pygamelib Documentation

Release 1.3.0

Arnaud Dupuis

Jan 23, 2024

CONTENTS (API REFERENCE):

pygamelib

CHAPTER ONE

FOREWORDS

Historically, this library was (and still is) used as a base to teach coding to kids from 6 to 15. It aims at giving an environment to new and learning developers (including kids) that let them focus on the algorithm instead of the lousy display or precise management.

It started as a very simple library with very little capabilities, but over time it became something more. To the point that it is now possible to make very decent terminal games with it.

So this is **obviously** still extremely simple compared to other game framework and it still does not have the pretention of being anything serious for real game developers. However, it can now be used by aspiring game developers for an introduction to 2D games development.

CHAPTER

TWO

INTRODUCTION

First of all, his module is exclusively compatible with python 3.6+.

The core concept is that writting a game mostly involve the `Game` object, the `Board` object and the derivatives of *board_items*.

More advanced game will use the *ui* module to create terminal user interfaces (or TUI) and the GFX *core* module to improve the graphics with `Sprite` and `Color`.

Here is an example of what the current version allow to build:

And a quick peak at the new features in the most recent version:

CHAPTER

THREE

TUTORIALS

Most tutorials to teach you how to use the library to build games are (or will be) on the wiki.

Tutorials that teach you how to expand the library are (or will be) centralized here.

The complete API documentation is referenced bellow.

3.1 actuators

This module contains the base classes for simple and advanced actuators. These classes are the base contract for actuators. If you wish to create your own one, you need to inherit from one of these base class.

3.1.1 Actuator

`class pygamelib.actuators.Actuator`(*parent:* BoardItem | *None*)

Bases: *`PglBaseObject`*

Actuator is the base class for all Actuators. It is mainly a contract class with some utility methods.

By default, all actuators are considered movement actuators. So the base class only require next_move() to be implemented.

Parameters

`parent` – the item parent.

`__init__`(*parent:* BoardItem | *None*)

The constructor take only one (positional) parameter: the parent object.

Important: The default state of ALL actuators is RUNNING. If you want your actuator to be in a different state (PAUSED for example), you have to do it yourself.

Methods

__init__(parent)	The constructor take only one (positional) parameter: the parent object.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load serialized data, create and returns a new actuator out of these data.
next_move()	That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the actuator state to PAUSED.
serialize()	Serializes the actuator and returns it as a dict.
start()	Set the actuator state to RUNNING.
stop()	Set the actuator state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

load(*data: dict*) → *Actuator*

Load serialized data, create and returns a new actuator out of these data.

That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

next_move() → *Vector2D* | int

That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

Returns
Vector2D | int

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serializes the actuator and returns it as a dict.

That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

start()

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

stop()

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.1.2 Behavioral

class pygamelib.actuators.**Behavioral**(*parent: BoardItem | None*)

Bases: *Actuator*

The behavioral actuator is inheriting from Actuator and is adding a next_action() method. The actual actions are left to the actuator that implements Behavioral.

Parameters
parent – the item parent.

__init__(*parent: BoardItem | None*)

The constructor simply construct an Actuator. It takes on positional parameter: the parent object.

Methods

__init__(parent)	The constructor simply construct an Actuator.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load serialized data, create and returns a new actuator out of these data.
next_action()	That method needs to be implemented by all behavioral actuators or a NotImplementedError exception will be raised.
next_move()	That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the actuator state to PAUSED.
serialize()	Serializes the actuator and returns it as a dict.
start()	Set the actuator state to RUNNING.
stop()	Set the actuator state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
`observer` (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
`observer` (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- `subject` (*PglBaseObject*) – The object that has changed.
- `attribute` (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- `value` (*Any*) – The new value of the attribute. This can be None.

load(*data: dict*) → *Actuator*

Load serialized data, create and returns a new actuator out of these data.

That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

next_action()

That method needs to be implemented by all behavioral actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

next_move() → *Vector2D* | int

That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

Returns
`Vector2D` | int

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serializes the actuator and returns it as a dict.

That method needs to be implemented by all actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

start()

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

stop()

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.1.3 PathActuator

class pygamelib.actuators.**PathActuator**(*path: List[int]* | *None = None*, *parent:* BoardItem | *None = None*)

Bases: *Actuator*

The path actuator is a subclass of *Actuator*. The move inside the function next_move depends on path and index. If the state is not running it returns None otherwise it increments the index & then, further compares the index with length of the path. If they both are same then, index is set to value zero and the move is returned back.

Parameters

- **path** (*list*) – A list of paths.
- **parent** (pygamelib.board_items.BoardItem) – The parent object to actuate.

__init__(*path: List[int]* | *None = None*, *parent:* BoardItem | *None = None*)

The constructor take only one (positional) parameter: the parent object.

Important: The default state of ALL actuators is RUNNING. If you want your actuator to be in a different state (PAUSED for example), you have to do it yourself.

Methods

__init__([path, parent])	The constructor take only one (positional) parameter: the parent object.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new PathActuator out of it.
next_move()	Return the movement based on current index
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the actuator state to PAUSED.
serialize()	Return a dictionary with all the attributes of this object.
set_path(path)	Defines a new path
start()	Set the actuator state to RUNNING.
stop()	Set the actuator state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

`attach`(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
`observer` (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data: dict*) → *PathActuator*

Load data and create a new PathActuator out of it.

Parameters
data (*dict*) – Data to create a new actuator (usually generated by `serialize()`)

Returns
A new actuator.

Return type
PathActuator

Example:

```
path_actuator = PathActuator.load(actuator_data)
```

next_move() → int

Return the movement based on current index

The movement is selected from path if state is RUNNING, otherwise it returns NO_DIR from the `constants` module. When state is RUNNING, the movement is selected before incrementing the index by 1. When the index equal the length of path, the index should return back to 0.

Returns
The next movement

Return type

int | *pygamelib.constants.Direction.NO_DIR*

Example:

```
path_actuator.next_move()
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns

A dictionary with all the attributes of this object.

Return type

dict

set_path(*path: List[int]*)

Defines a new path

This will also reset the index back to 0.

Parameters
path (*list*) – A list of movements.

Example:

```
path_actuator.set_path([Direction.UP,Direction.DOWN,Direction.LEFT,Direction.
↪RIGHT])
```

start()

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

stop()

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.1.4 PatrolActuator

class pygamelib.actuators.**PatrolActuator**(*path: List[int] | None = None*, *parent: BoardItem | None = None*)

Bases: *PathActuator*

The patrol actuator is a subclass of *PathActuator*. The move inside the function next_move depends on path and index and the mode. Once it reaches the end of the move list it will start cycling back to the beginning of the list. Once it reaches the beginning it will start moving forwards If the state is not running it returns None otherwise it increments the index & then, further compares the index with length of the path. If they both are same then, index is set to value zero and the move is returned back.

Parameters
path (*list*) – A list of directions.

__init__(*path: List[int] | None = None*, *parent: BoardItem | None = None*)

The constructor take only one (positional) parameter: the parent object.

Important: The default state of ALL actuators is RUNNING. If you want your actuator to be in a different state (PAUSED for example), you have to do it yourself.

Methods

__init__([path, parent])	The constructor take only one (positional) parameter: the parent object.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new PatrolActuator out of it.
next_move()	Return the movement based on current index
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the actuator state to PAUSED.
serialize()	Return a dictionary with all the attributes of this object.
set_path(path)	Defines a new path
start()	Set the actuator state to RUNNING.
stop()	Set the actuator state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
```

(continues on next page)

(continued from previous page)

```
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data: dict*) → *PatrolActuator*

Load data and create a new PatrolActuator out of it.

Parameters
data (*dict*) – Data to create a new actuator (usually generated by `serialize()`)

Returns
A new actuator.

Return type
PatrolActuator

Example:

```
patrol_actuator = PatrolActuator.load(actuator_data)
```

next_move() → int

Return the movement based on current index

The movement is selected from path if state is RUNNING, otherwise it returns NO_DIR from the `constants` module. When state is RUNNING, the movement is selected before incrementing the index by

1. When the index equals the length of path, the index should return back to 0 and the path list should be reversed before the next call.

Returns
The next movement

Return type
int | *pygamelib.constants.Direction.NO_DIR*

Example:

```
patrol_actuator.next_move()
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_path(*path: List[int]*)

Defines a new path

This will also reset the index back to 0.

Parameters
path (*list*) – A list of movements.

Example:

```
path_actuator.set_path([Direction.UP,Direction.DOWN,Direction.LEFT,Direction.
↪RIGHT])
```

start()

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

stop()

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.1.5 PathFinder

class `pygamelib.actuators.`**`PathFinder`**(*game: engine.Game | None = None, actuated_object: board_items.BoardItem | None = None, circle_waypoints=True, parent: board_items.BoardItem | None = None, algorithm=Algorithm.BFS*)

Bases: *`Behavioral`*

Important: This module assume a one step movement. If you need more than one step, you will need to sub-class this module and re-implement next_waypoint().

This actuator is a bit different than the simple actuators (`SimpleActuators`) as it requires the knowledge of both the game object and the actuated object.

The constructor takes the following parameters:

Parameters

- **`game`** (`pygamelib.engine.Game`) – A reference to the instantiated game engine.
- **`actuated_object`** (`pygamelib.board_items.BoardItem`) – The object to actuate. Deprecated in favor of parent. Only kept for backward compatibility.
- **`parent`** (`pygamelib.board_items.BoardItem`) – The parent object to actuate.
- **`circle_waypoints`** (*`bool`*) – If True the next_waypoint() method is going to circle between the waypoints (when the last is visited, go back to the first)
- **`algorithm`** (*`constant`*) – ALGO_BFS - BFS, ALGO_ASTAR - AStar

`__init__`(*game: engine.Game | None = None, actuated_object: board_items.BoardItem | None = None, circle_waypoints=True, parent: board_items.BoardItem | None = None, algorithm=Algorithm.BFS*)

The constructor simply construct an Actuator. It takes on positional parameter: the parent object.

Methods

__init__([game, actuated_object, ...])	The constructor simply construct an Actuator.
add_waypoint(row, column)	Add a waypoint to the list of waypoints.
attach(observer)	Attach an observer to this instance.
clear_waypoints()	Empty the waypoints stack.
current_path()	This method simply return a copy of the current path of the actuator.
current_waypoint()	Return the currently active waypoint.
detach(observer)	Detach an observer from this instance.
find_path()	Find a path to the destination.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new PathFinder out of it.
next_action()	That method needs to be implemented by all behavioral actuators or a NotImplementedError exception will be raised.
next_move()	This method return the next move calculated by this actuator.
next_waypoint()	Return the next active waypoint.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the actuator state to PAUSED.
remove_waypoint(row, column)	Remove a waypoint from the stack.
serialize()	Return a dictionary with all the attributes of this object.
set_destination([row, column])	Set the targeted destination.
start()	Set the actuator state to RUNNING.
stop()	Set the actuator state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

add_waypoint(*row: int*, *column: int*)

Add a waypoint to the list of waypoints.

Waypoints are used one after the other on a FIFO basis (First In, First Out).

If not destination (i.e destination == (None, None)) have been set yet, that method sets it.

Parameters

- **row** (*int*) – The "row" part of the waypoint's coordinate.
- **column** – The "column" part of the waypoint's coordinate.

Raises

PglInvalidTypeException – If any of the parameters is not an int.

Example:

```
pf = PathFinder(game=mygame, actuated_object=npc1)
pf.add_waypoint(3,5)
pf.add_waypoint(12,15)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

clear_waypoints()

Empty the waypoints stack.

Example:

```
pf.clear_waypoints()
```

current_path() → List[Tuple[int, int]]

This method simply return a copy of the current path of the actuator.

The current path is to be understood as: the list of positions still remaining. All positions that have already been gone through are removed from the stack.

Important: A copy of the path is returned for every call to that function so be wary of the performances impact.

Example:

```
mykillernpc.actuator = PathFinder(
                        game=mygame,
                        actuated_object=mykillernpc
                    )
mykillernpc.actuator.set_destination(
                        mygame.player.pos[0],
                        mygame.player.pos[1]
                    )
mykillernpc.actuator.find_path()
for i in mykillernpc.actuator.current_path():
    print(i)
```

current_waypoint() → Tuple[int | None, int | None]

Return the currently active waypoint.

If no waypoint have been added, this function return None.

Returns
Either a None tuple or the current waypoint.

Return type
A None tuple or a tuple of integer.

Example:

```
(row,column) = pf.current_waypoint()
pf.set_destination(row,column)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

find_path() → List[Tuple[int, int]]

Find a path to the destination.

Destination (PathFinder.destination) has to be set beforehand.

Example:

```
mykillernpc.actuator = PathFinder(
        game=mygame, actuated_object=mykillernpc
    )
mykillernpc.actuator.set_destination(
        mygame.player.pos[0], mygame.player.pos[1]
    )
mykillernpc.actuator.find_path()
```

Warning: PathFinder.destination is a tuple! Please use PathFinder.set_destination(x,y) to avoid problems.

Path Finding Algorithm Description:

Breadth First Search: This method implements a Breadth First Search algorithm (Wikipedia: BFS) to find the shortest path to destination.

A* Search: This method implements a A* Search algorithm (Wikipedia: A*) to find the shortest path to destination.

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data: dict*) → *PathFinder*

Load data and create a new PathFinder out of it.

Parameters
data (*dict*) – Data to create a new actuator (usually generated by *serialize()*)

Returns
A new actuator.

Return type
PathFinder

Example:

```
path_finder = PathFinder.load(actuator_data)
```

next_action()

That method needs to be implemented by all behavioral actuators or a NotImplementedError exception will be raised.

Raises
NotImplementedError

next_move() → int

This method return the next move calculated by this actuator.

In the case of this PathFinder actuator, next move does the following:

- If the destination is not set return NO_DIR (see *constants*) - If the destination is set, but the path is empty and actuated object's position is different from destination: call *find_path()*
- Look at the current waypoint, if the actuated object is not at that position return a direction from the *constants* module. The direction is calculated from the difference between actuated object's position and waypoint's position.
- If the actuated object is at the waypoint position, then call next_waypoint(), set the destination and return a direction. In this case, also call *find_path()*.
- In any case, if there is no more waypoints in the path this method returns NO_DIR (see *constants*)

Example:

```
seeker = NPC(model=graphics.Models.SKULL)
seeker.actuator = PathFinder(game=mygame,actuated_object=seeker)
while True:
    seeker.actuator.set_destination(mygame.player.pos[0],mygame.player.pos[1])
    # next_move() will call find_path() for us.
    next_move = seeker.actuator.next_move()
    if next_move == Direction.NO_DIR:
        seeker.actuator.set_destination(mygame.player.pos[0],mygame.player.
↪pos[1])
    else:
        mygame.current_board().move(seeker,next_move,1)
```

next_waypoint() → Tuple[int | None, int | None]

Return the next active waypoint.

If no waypoint have been added, this function return None. If there is no more waypoint in the stack:

- if PathFinder.circle_waypoints is True this function reset the waypoints stack and return the first one.
- else, return None.

Returns
Either a None tuple or the next waypoint.

Return type
A None tuple or a tuple of integer.

Example:

```
pf.circle_waypoints = True
(row,column) = pf.next_waypoint()
pf.set_destination(row,column)
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

remove_waypoint(*row: int*, *column: int*)

Remove a waypoint from the stack.

This method removes the first occurrence of a waypoint in the stack.

If the waypoint cannot be found, it raises a ValueError exception. If the row and column parameters are not int, an PglInvalidTypeException is raised.

Parameters

- **row** (*int*) – The "row" part of the waypoint's coordinate.
- **column** – The "column" part of the waypoint's coordinate.

Raises

- *PglInvalidTypeException* – If any of the parameters is not an int.
- **ValueError** – If the waypoint is not found in the stack.

Example:

```
path_finder.remove_waypoint(2,5)
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns

A dictionary with all the attributes of this object.

Return type

dict

set_destination(*row: int = 0*, *column: int = 0*)

Set the targeted destination.

Parameters

- **row** (*int*) – "row" coordinate on the board grid
- **column** (*int*) – "column" coordinate on the board grid

Raises

PglInvalidTypeException – if row or column are not int.

Example:

```
mykillernpc.actuator.set_destination(
    mygame.player.pos[0], mygame.player.pos[1]
)
```

`start()`

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

`stop()`

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

`store_screen_position`(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.1.6 RandomActuator

class pygamelib.actuators.**RandomActuator**(*moveset: List[*Vector2D *| int] | None = None*, *parent:* BoardItem *| None = None*)

Bases: *Actuator*

A class that implements a random choice of movement.

The random actuator is a subclass of *Actuator*. It is simply implementing a random choice in a predefined move set.

Parameters

- **moveset** (*list*) – A list of movements.
- **parent** (pygamelib.board_items.BoardItem) – The parent object to actuate.

__init__(*moveset: List[*Vector2D *| int] | None = None*, *parent:* BoardItem *| None = None*)

The constructor take only one (positional) parameter: the parent object.

Important: The default state of ALL actuators is RUNNING. If you want your actuator to be in a different state (PAUSED for example), you have to do it yourself.

Methods

__init__([moveset, parent])	The constructor take only one (positional) parameter: the parent object.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new RandomActuator out of it.
next_move()	Return a randomly selected movement
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the actuator state to PAUSED.
serialize()	Return a dictionary with all the attributes of this object.
start()	Set the actuator state to RUNNING.
stop()	Set the actuator state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

moveset	Return the moveset.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
`observer` (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- `subject` (*PglBaseObject*) – The object that has changed.
- `attribute` (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- `value` (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data: dict*) → *RandomActuator*

Load data and create a new RandomActuator out of it.

Parameters
`data` (*dict*) – Data to create a new actuator (usually generated by `serialize()`)

Returns
A new actuator.

Return type
RandomActuator

Example:

```
npc2.actuator = actuators.RandomActuator.load( npc1.actuator.serialize() )
```

property moveset: List[*Vector2D* | int]

Return the moveset.

Returns
The moveset.

Return type
list

next_move() → *Vector2D* | int

Return a randomly selected movement

The movement is randomly selected from moveset if state is RUNNING, otherwise it returns NO_DIR from the `constants` module.

Returns
The next movement `Vector2D` | int

Example:

```
random_actuator.next_move()
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

start()

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

stop()

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.1.7 UnidirectionalActuator

class pygamelib.actuators.**UnidirectionalActuator**(*direction: int = Direction.RIGHT*, *parent: BoardItem | None = None*)

Bases: *Actuator*

A class that implements a single movement.

The unidirectional actuator is a subclass of *Actuator*. It is simply implementing a mono directional movement. It is primarily target at projectiles.

Parameters

- **direction** (*int*) – A single direction from the Constants module.
- **parent** (pygamelib.board_items.BoardItem) – The parent object to actuate.

`__init__`(*direction: int = Direction.RIGHT*, *parent:* BoardItem | *None = None*)

The constructor take only one (positional) parameter: the parent object.

Important: The default state of ALL actuators is RUNNING. If you want your actuator to be in a different state (PAUSED for example), you have to do it yourself.

Methods

`__init__`([direction, parent])	The constructor take only one (positional) parameter: the parent object.
`attach`(observer)	Attach an observer to this instance.
`detach`(observer)	Detach an observer from this instance.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load data and create a new UnidirectionalActuator out of it.
`next_move`()	Return the direction.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`pause`()	Set the actuator state to PAUSED.
`serialize`()	Return a dictionary with all the attributes of this object.
`start`()	Set the actuator state to RUNNING.
`stop`()	Set the actuator state to STOPPED.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

`screen_column`	A property to get/set the screen column.
`screen_row`	A property to get/set the screen row.

`attach`(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

`observer` (*`PglBaseObject`*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
```

(continues on next page)

(continued from previous page)

```
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
`observer` (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- `subject` (*PglBaseObject*) – The object that has changed.
- `attribute` (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- `value` (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data: dict*) → *UnidirectionalActuator*

Load data and create a new UnidirectionalActuator out of it.

Parameters
`data` (*dict*) – Data to create a new actuator (usually generated by `serialize()`)

Returns
A new actuator.

Return type
UnidirectionalActuator

Example:

```
unidir_actuator = UnidirectionalActuator.load(actuator_data)
```

next_move() → int

Return the direction.

The movement is always direction if state is RUNNING, otherwise it returns NO_DIR from the `constants` module.

Returns
The next movement

Return type
int | `pygamelib.Direction.NO_DIR`

Example:

```
unidirectional_actuator.next_move()
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

start()

Set the actuator state to RUNNING.

If the actuator state is not RUNNING, actuators' next_move() function (and all derivatives) should not return anything.

Example:

```
mygame.start()
```

stop()

Set the actuator state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.2 assets

The assets sub-module holds all the classes that are adding features without being core features. The graphics module is a good example of that: it is cool to have and provides a nice default set of assets to build games. But the library can work without it.

3.2.1 graphics

Important: The Graphics module was introduced in version 1.1.0.

The Graphics module hold many variables that aims at simplifying the use of unicode characters in the game development process.

This module also import colorama. All styling features are accessible through:

- Graphics.Fore for Foreground colors.
- Graphics.Back for Background colors.
- Graphics.Style for styling options.

For convenience, the different entities are scattered in grouping classes:

- All emojis are in the Models class.
- The UI/box drawings are grouped into the BoxDrawings class.
- The block glyphs are in the Blocks class.
- The geometric shapes are in the GeometricShapes class.

This modules defines a couple of colored squares and rectangles that should displays correctly in all terminals.

These are kept for legacy purpose (I personally have a lot of kids that are still using it), but for anyone starting fresh, it is better to use the <color>_rect() and <color>_square() static methods of the `Sprixel` class. Particularly if you are going to use them as background for your Board.

Colored rectangles:

- WHITE_RECT
- BLUE_RECT
- RED_RECT
- MAGENTA_RECT
- GREEN_RECT
- YELLOW_RECT
- BLACK_RECT
- CYAN_RECT

Then colored squares:

- WHITE_SQUARE
- MAGENTA_SQUARE
- GREEN_SQUARE
- RED_SQUARE
- BLUE_SQUARE
- YELLOW_SQUARE
- BLACK_SQUARE
- CYAN_SQUARE

And finally an example of composition of rectangles to make different colored squares:

- RED_BLUE_SQUARE = RED_RECT+BLUE_RECT
- YELLOW_CYAN_SQUARE = YELLOW_RECT+CYAN_RECT

The Graphics module contains the following classes:

Blocks

class pygamelib.assets.graphics.**Blocks**

Bases: `object`

Block elements (unicode)

Here is the list of supported glyphs:

- UPPER_HALF_BLOCK =
- LOWER_ONE_EIGHTH_BLOCK =
- LOWER_ONE_QUARTER_BLOCK =
- LOWER_THREE_EIGHTHS_BLOCK =
- LOWER_HALF_BLOCK =
- LOWER_FIVE_EIGHTHS_BLOCK =
- LOWER_THREE_QUARTERS_BLOCK =
- LOWER_SEVEN_EIGHTHS_BLOCK =
- FULL_BLOCK =
- LEFT_SEVEN_EIGHTHS_BLOCK =
- LEFT_THREE_QUARTERS_BLOCK =
- LEFT_FIVE_EIGHTHS_BLOCK =
- LEFT_HALF_BLOCK =
- LEFT_THREE_EIGHTHS_BLOCK =
- LEFT_ONE_QUARTER_BLOCK =
- LEFT_ONE_EIGHTH_BLOCK =
- RIGHT_HALF_BLOCK =
- LIGHT_SHADE =
- MEDIUM_SHADE =
- DARK_SHADE =
- UPPER_ONE_EIGHTH_BLOCK =
- RIGHT_ONE_EIGHTH_BLOCK =
- QUADRANT_LOWER_LEFT =
- QUADRANT_LOWER_RIGHT =
- QUADRANT_UPPER_LEFT =
- QUADRANT_UPPER_LEFT_AND_LOWER_LEFT_AND_LOWER_RIGHT =
- QUADRANT_UPPER_LEFT_AND_LOWER_RIGHT =
- QUADRANT_UPPER_LEFT_AND_UPPER_RIGHT_AND_LOWER_LEFT =
- QUADRANT_UPPER_LEFT_AND_UPPER_RIGHT_AND_LOWER_RIGHT =
- QUADRANT_UPPER_RIGHT =
- QUADRANT_UPPER_RIGHT_AND_LOWER_LEFT =

- QUADRANT_UPPER_RIGHT_AND_LOWER_LEFT_AND_LOWER_RIGHT =

__init__()

Methods

__init__()	

Attributes

DARK_SHADE	
FULL_BLOCK	
LEFT_FIVE_EIGHTHS_BLOCK	
LEFT_HALF_BLOCK	
LEFT_ONE_EIGHTH_BLOCK	
LEFT_ONE_QUARTER_BLOCK	
LEFT_SEVEN_EIGHTHS_BLOCK	
LEFT_THREE_EIGHTHS_BLOCK	
LEFT_THREE_QUARTERS_BLOCK	
LIGHT_SHADE	
LOWER_FIVE_EIGHTHS_BLOCK	
LOWER_HALF_BLOCK	
LOWER_ONE_EIGHTH_BLOCK	
LOWER_ONE_QUARTER_BLOCK	
LOWER_SEVEN_EIGHTHS_BLOCK	
LOWER_THREE_EIGHTHS_BLOCK	
LOWER_THREE_QUARTERS_BLOCK	
MEDIUM_SHADE	
QUADRANT_LOWER_LEFT	

continues on next page

Table 1 – continued from previous page

QUADRANT_LOWER_RIGHT
QUADRANT_UPPER_LEFT
QUADRANT_UPPER_LEFT_AND_LOWER_LEFT_AND_LO
QUADRANT_UPPER_LEFT_AND_LOWER_RIGHT
QUADRANT_UPPER_LEFT_AND_UPPER_RIGHT_AND_L
QUADRANT_UPPER_LEFT_AND_UPPER_RIGHT_AND_L
QUADRANT_UPPER_RIGHT
QUADRANT_UPPER_RIGHT_AND_LOWER_LEFT
QUADRANT_UPPER_RIGHT_AND_LOWER_LEFT_AND_L
RIGHT_HALF_BLOCK
RIGHT_ONE_EIGHTH_BLOCK
UPPER_HALF_BLOCK
UPPER_ONE_EIGHTH_BLOCK

DARK_SHADE = ''

FULL_BLOCK = ''

LEFT_FIVE_EIGHTHS_BLOCK = ''

LEFT_HALF_BLOCK = ''

LEFT_ONE_EIGHTH_BLOCK = ''

LEFT_ONE_QUARTER_BLOCK = ''

LEFT_SEVEN_EIGHTHS_BLOCK = ''

LEFT_THREE_EIGHTHS_BLOCK = ''

LEFT_THREE_QUARTERS_BLOCK = ''

LIGHT_SHADE = ''

LOWER_FIVE_EIGHTHS_BLOCK = ''

LOWER_HALF_BLOCK = ''

LOWER_ONE_EIGHTH_BLOCK = ''

LOWER_ONE_QUARTER_BLOCK = ''

LOWER_SEVEN_EIGHTHS_BLOCK = ''

LOWER_THREE_EIGHTHS_BLOCK = ''

LOWER_THREE_QUARTERS_BLOCK = ''

MEDIUM_SHADE = ''

QUADRANT_LOWER_LEFT = ''

QUADRANT_LOWER_RIGHT = ''

QUADRANT_UPPER_LEFT = ''

QUADRANT_UPPER_LEFT_AND_LOWER_LEFT_AND_LOWER_RIGHT = ''

QUADRANT_UPPER_LEFT_AND_LOWER_RIGHT = ''

QUADRANT_UPPER_LEFT_AND_UPPER_RIGHT_AND_LOWER_LEFT = ''

QUADRANT_UPPER_LEFT_AND_UPPER_RIGHT_AND_LOWER_RIGHT = ''

QUADRANT_UPPER_RIGHT = ''

QUADRANT_UPPER_RIGHT_AND_LOWER_LEFT = ''

QUADRANT_UPPER_RIGHT_AND_LOWER_LEFT_AND_LOWER_RIGHT = ''

RIGHT_HALF_BLOCK = ''

RIGHT_ONE_EIGHTH_BLOCK = ''

UPPER_HALF_BLOCK = ''

UPPER_ONE_EIGHTH_BLOCK = ''

BoxDrawings

class pygamelib.assets.graphics.**BoxDrawings**

Bases: object

Box drawing elements (unicode)

Here is the list of supported glyphs:

- LIGHT_HORIZONTAL = –
- HEAVY_HORIZONTAL =
- LIGHT_VERTICAL = │
- HEAVY_VERTICAL =
- LIGHT_TRIPLE_DASH_HORIZONTAL =
- HEAVY_TRIPLE_DASH_HORIZONTAL =
- LIGHT_TRIPLE_DASH_VERTICAL =
- HEAVY_TRIPLE_DASH_VERTICAL =
- LIGHT_QUADRUPLE_DASH_HORIZONTAL =

- HEAVY_QUADRUPLE_DASH_HORIZONTAL =
- LIGHT_QUADRUPLE_DASH_VERTICAL =
- HEAVY_QUADRUPLE_DASH_VERTICAL =
- LIGHT_DOWN_AND_RIGHT =
- DOWN_LIGHT_AND_RIGHT_HEAVY =
- DOWN_HEAVY_AND_RIGHT_LIGHT =
- HEAVY_DOWN_AND_RIGHT =
- LIGHT_DOWN_AND_LEFT =
- DOWN_LIGHT_AND_LEFT_HEAVY =
- DOWN_HEAVY_AND_LEFT_LIGHT =
- HEAVY_DOWN_AND_LEFT =
- LIGHT_UP_AND_RIGHT = └
- UP_LIGHT_AND_RIGHT_HEAVY =
- UP_HEAVY_AND_RIGHT_LIGHT =
- HEAVY_UP_AND_RIGHT =
- LIGHT_UP_AND_LEFT =
- UP_LIGHT_AND_LEFT_HEAVY =
- UP_HEAVY_AND_LEFT_LIGHT =
- HEAVY_UP_AND_LEFT =
- LIGHT_VERTICAL_AND_RIGHT = ├
- VERTICAL_LIGHT_AND_RIGHT_HEAVY =
- UP_HEAVY_AND_RIGHT_DOWN_LIGHT =
- DOWN_HEAVY_AND_RIGHT_UP_LIGHT =
- VERTICAL_HEAVY_AND_RIGHT_LIGHT =
- DOWN_LIGHT_AND_RIGHT_UP_HEAVY =
- UP_LIGHT_AND_RIGHT_DOWN_HEAVY =
- HEAVY_VERTICAL_AND_RIGHT =
- LIGHT_VERTICAL_AND_LEFT =
- VERTICAL_LIGHT_AND_LEFT_HEAVY =
- UP_HEAVY_AND_LEFT_DOWN_LIGHT =
- DOWN_HEAVY_AND_LEFT_UP_LIGHT =
- VERTICAL_HEAVY_AND_LEFT_LIGHT =
- DOWN_LIGHT_AND_LEFT_UP_HEAVY =
- UP_LIGHT_AND_LEFT_DOWN_HEAVY =
- HEAVY_VERTICAL_AND_LEFT =
- LIGHT_DOWN_AND_HORIZONTAL =

- LEFT_HEAVY_AND_RIGHT_DOWN_LIGHT =
- RIGHT_HEAVY_AND_LEFT_DOWN_LIGHT =
- DOWN_LIGHT_AND_HORIZONTAL_HEAVY =
- DOWN_HEAVY_AND_HORIZONTAL_LIGHT =
- RIGHT_LIGHT_AND_LEFT_DOWN_HEAVY =
- LEFT_LIGHT_AND_RIGHT_DOWN_HEAVY =
- HEAVY_DOWN_AND_HORIZONTAL =
- LIGHT_UP_AND_HORIZONTAL =
- LEFT_HEAVY_AND_RIGHT_UP_LIGHT =
- RIGHT_HEAVY_AND_LEFT_UP_LIGHT =
- UP_LIGHT_AND_HORIZONTAL_HEAVY =
- UP_HEAVY_AND_HORIZONTAL_LIGHT =
- RIGHT_LIGHT_AND_LEFT_UP_HEAVY =
- LEFT_LIGHT_AND_RIGHT_UP_HEAVY =
- HEAVY_UP_AND_HORIZONTAL =
- LIGHT_VERTICAL_AND_HORIZONTAL =
- LEFT_HEAVY_AND_RIGHT_VERTICAL_LIGHT =
- RIGHT_HEAVY_AND_LEFT_VERTICAL_LIGHT =
- VERTICAL_LIGHT_AND_HORIZONTAL_HEAVY =
- UP_HEAVY_AND_DOWN_HORIZONTAL_LIGHT =
- DOWN_HEAVY_AND_UP_HORIZONTAL_LIGHT =
- VERTICAL_HEAVY_AND_HORIZONTAL_LIGHT =
- LEFT_UP_HEAVY_AND_RIGHT_DOWN_LIGHT =
- RIGHT_UP_HEAVY_AND_LEFT_DOWN_LIGHT =
- LEFT_DOWN_HEAVY_AND_RIGHT_UP_LIGHT =
- RIGHT_DOWN_HEAVY_AND_LEFT_UP_LIGHT =
- DOWN_LIGHT_AND_UP_HORIZONTAL_HEAVY =
- UP_LIGHT_AND_DOWN_HORIZONTAL_HEAVY =
- RIGHT_LIGHT_AND_LEFT_VERTICAL_HEAVY =
- LEFT_LIGHT_AND_RIGHT_VERTICAL_HEAVY =
- HEAVY_VERTICAL_AND_HORIZONTAL =
- LIGHT_DOUBLE_DASH_HORIZONTAL =
- HEAVY_DOUBLE_DASH_HORIZONTAL =
- LIGHT_DOUBLE_DASH_VERTICAL =
- HEAVY_DOUBLE_DASH_VERTICAL =
- DOUBLE_HORIZONTAL =

- DOUBLE_VERTICAL =
- DOWN_SINGLE_AND_RIGHT_DOUBLE =
- DOWN_DOUBLE_AND_RIGHT_SINGLE =
- DOUBLE_DOWN_AND_RIGHT =
- DOWN_SINGLE_AND_LEFT_DOUBLE =
- DOWN_DOUBLE_AND_LEFT_SINGLE =
- DOUBLE_DOWN_AND_LEFT =
- UP_SINGLE_AND_RIGHT_DOUBLE =
- UP_DOUBLE_AND_RIGHT_SINGLE =
- DOUBLE_UP_AND_RIGHT =
- UP_SINGLE_AND_LEFT_DOUBLE =
- UP_DOUBLE_AND_LEFT_SINGLE =
- DOUBLE_UP_AND_LEFT =
- VERTICAL_SINGLE_AND_RIGHT_DOUBLE =
- VERTICAL_DOUBLE_AND_RIGHT_SINGLE =
- DOUBLE_VERTICAL_AND_RIGHT =
- VERTICAL_SINGLE_AND_LEFT_DOUBLE =
- VERTICAL_DOUBLE_AND_LEFT_SINGLE =
- DOUBLE_VERTICAL_AND_LEFT =
- DOWN_SINGLE_AND_HORIZONTAL_DOUBLE =
- DOWN_DOUBLE_AND_HORIZONTAL_SINGLE =
- DOUBLE_DOWN_AND_HORIZONTAL =
- UP_SINGLE_AND_HORIZONTAL_DOUBLE =
- UP_DOUBLE_AND_HORIZONTAL_SINGLE =
- DOUBLE_UP_AND_HORIZONTAL =
- VERTICAL_SINGLE_AND_HORIZONTAL_DOUBLE =
- VERTICAL_DOUBLE_AND_HORIZONTAL_SINGLE =
- DOUBLE_VERTICAL_AND_HORIZONTAL =
- LIGHT_ARC_DOWN_AND_RIGHT =
- LIGHT_ARC_DOWN_AND_LEFT =
- LIGHT_ARC_UP_AND_LEFT =
- LIGHT_ARC_UP_AND_RIGHT =
- LIGHT_DIAGONAL_UPPER_RIGHT_TO_LOWER_LEFT =
- LIGHT_DIAGONAL_UPPER_LEFT_TO_LOWER_RIGHT = \
- LIGHT_DIAGONAL_CROSS =
- LIGHT_LEFT =

- LIGHT_UP =
- LIGHT_RIGHT =
- LIGHT_DOWN =
- HEAVY_LEFT =
- HEAVY_UP =
- HEAVY_RIGHT =
- HEAVY_DOWN =
- LIGHT_LEFT_AND_HEAVY_RIGHT =
- LIGHT_UP_AND_HEAVY_DOWN =
- HEAVY_LEFT_AND_LIGHT_RIGHT =
- HEAVY_UP_AND_LIGHT_DOWN =

`__init__()`

Methods

__init__()

Attributes

DOUBLE_DOWN_AND_HORIZONTAL
DOUBLE_DOWN_AND_LEFT
DOUBLE_DOWN_AND_RIGHT
DOUBLE_HORIZONTAL
DOUBLE_UP_AND_HORIZONTAL
DOUBLE_UP_AND_LEFT
DOUBLE_UP_AND_RIGHT
DOUBLE_VERTICAL
DOUBLE_VERTICAL_AND_HORIZONTAL
DOUBLE_VERTICAL_AND_LEFT
DOUBLE_VERTICAL_AND_RIGHT

continues on next page

Table 2 – continued from previous page

DOWN_DOUBLE_AND_HORIZONTAL_SINGLE
DOWN_DOUBLE_AND_LEFT_SINGLE
DOWN_DOUBLE_AND_RIGHT_SINGLE
DOWN_HEAVY_AND_HORIZONTAL_LIGHT
DOWN_HEAVY_AND_LEFT_LIGHT
DOWN_HEAVY_AND_LEFT_UP_LIGHT
DOWN_HEAVY_AND_RIGHT_LIGHT
DOWN_HEAVY_AND_RIGHT_UP_LIGHT
DOWN_HEAVY_AND_UP_HORIZONTAL_LIGHT
DOWN_LIGHT_AND_HORIZONTAL_HEAVY
DOWN_LIGHT_AND_LEFT_HEAVY
DOWN_LIGHT_AND_LEFT_UP_HEAVY
DOWN_LIGHT_AND_RIGHT_HEAVY
DOWN_LIGHT_AND_RIGHT_UP_HEAVY
DOWN_LIGHT_AND_UP_HORIZONTAL_HEAVY
DOWN_SINGLE_AND_HORIZONTAL_DOUBLE
DOWN_SINGLE_AND_LEFT_DOUBLE
DOWN_SINGLE_AND_RIGHT_DOUBLE
HEAVY_DOUBLE_DASH_HORIZONTAL
HEAVY_DOUBLE_DASH_VERTICAL
HEAVY_DOWN
HEAVY_DOWN_AND_HORIZONTAL
HEAVY_DOWN_AND_LEFT
HEAVY_DOWN_AND_RIGHT
HEAVY_HORIZONTAL
HEAVY_LEFT

continues on next page

Table 2 – continued from previous page

HEAVY_LEFT_AND_LIGHT_RIGHT
HEAVY_QUADRUPLE_DASH_HORIZONTAL
HEAVY_QUADRUPLE_DASH_VERTICAL
HEAVY_RIGHT
HEAVY_TRIPLE_DASH_HORIZONTAL
HEAVY_TRIPLE_DASH_VERTICAL
HEAVY_UP
HEAVY_UP_AND_HORIZONTAL
HEAVY_UP_AND_LEFT
HEAVY_UP_AND_LIGHT_DOWN
HEAVY_UP_AND_RIGHT
HEAVY_VERTICAL
HEAVY_VERTICAL_AND_HORIZONTAL
HEAVY_VERTICAL_AND_LEFT
HEAVY_VERTICAL_AND_RIGHT
LEFT_DOWN_HEAVY_AND_RIGHT_UP_LIGHT
LEFT_HEAVY_AND_RIGHT_DOWN_LIGHT
LEFT_HEAVY_AND_RIGHT_UP_LIGHT
LEFT_HEAVY_AND_RIGHT_VERTICAL_LIGHT
LEFT_LIGHT_AND_RIGHT_DOWN_HEAVY
LEFT_LIGHT_AND_RIGHT_UP_HEAVY
LEFT_LIGHT_AND_RIGHT_VERTICAL_HEAVY
LEFT_UP_HEAVY_AND_RIGHT_DOWN_LIGHT
LIGHT_ARC_DOWN_AND_LEFT
LIGHT_ARC_DOWN_AND_RIGHT
LIGHT_ARC_UP_AND_LEFT

continues on next page

Table 2 – continued from previous page

LIGHT_ARC_UP_AND_RIGHT
LIGHT_DIAGONAL_CROSS
LIGHT_DIAGONAL_UPPER_LEFT_TO_LOWER_RIGHT
LIGHT_DIAGONAL_UPPER_RIGHT_TO_LOWER_LEFT
LIGHT_DOUBLE_DASH_HORIZONTAL
LIGHT_DOUBLE_DASH_VERTICAL
LIGHT_DOWN
LIGHT_DOWN_AND_HORIZONTAL
LIGHT_DOWN_AND_LEFT
LIGHT_DOWN_AND_RIGHT
LIGHT_HORIZONTAL
LIGHT_LEFT
LIGHT_LEFT_AND_HEAVY_RIGHT
LIGHT_QUADRUPLE_DASH_HORIZONTAL
LIGHT_QUADRUPLE_DASH_VERTICAL
LIGHT_RIGHT
LIGHT_TRIPLE_DASH_HORIZONTAL
LIGHT_TRIPLE_DASH_VERTICAL
LIGHT_UP
LIGHT_UP_AND_HEAVY_DOWN
LIGHT_UP_AND_HORIZONTAL
LIGHT_UP_AND_LEFT
LIGHT_UP_AND_RIGHT
LIGHT_VERTICAL
LIGHT_VERTICAL_AND_HORIZONTAL
LIGHT_VERTICAL_AND_LEFT

continues on next page

Table 2 – continued from previous page

LIGHT_VERTICAL_AND_RIGHT
RIGHT_DOWN_HEAVY_AND_LEFT_UP_LIGHT
RIGHT_HEAVY_AND_LEFT_DOWN_LIGHT
RIGHT_HEAVY_AND_LEFT_UP_LIGHT
RIGHT_HEAVY_AND_LEFT_VERTICAL_LIGHT
RIGHT_LIGHT_AND_LEFT_DOWN_HEAVY
RIGHT_LIGHT_AND_LEFT_UP_HEAVY
RIGHT_LIGHT_AND_LEFT_VERTICAL_HEAVY
RIGHT_UP_HEAVY_AND_LEFT_DOWN_LIGHT
UP_DOUBLE_AND_HORIZONTAL_SINGLE
UP_DOUBLE_AND_LEFT_SINGLE
UP_DOUBLE_AND_RIGHT_SINGLE
UP_HEAVY_AND_DOWN_HORIZONTAL_LIGHT
UP_HEAVY_AND_HORIZONTAL_LIGHT
UP_HEAVY_AND_LEFT_DOWN_LIGHT
UP_HEAVY_AND_LEFT_LIGHT
UP_HEAVY_AND_RIGHT_DOWN_LIGHT
UP_HEAVY_AND_RIGHT_LIGHT
UP_LIGHT_AND_DOWN_HORIZONTAL_HEAVY
UP_LIGHT_AND_HORIZONTAL_HEAVY
UP_LIGHT_AND_LEFT_DOWN_HEAVY
UP_LIGHT_AND_LEFT_HEAVY
UP_LIGHT_AND_RIGHT_DOWN_HEAVY
UP_LIGHT_AND_RIGHT_HEAVY
UP_SINGLE_AND_HORIZONTAL_DOUBLE
UP_SINGLE_AND_LEFT_DOUBLE

continues on next page

Table 2 – continued from previous page

UP_SINGLE_AND_RIGHT_DOUBLE
VERTICAL_DOUBLE_AND_HORIZONTAL_SINGLE
VERTICAL_DOUBLE_AND_LEFT_SINGLE
VERTICAL_DOUBLE_AND_RIGHT_SINGLE
VERTICAL_HEAVY_AND_HORIZONTAL_LIGHT
VERTICAL_HEAVY_AND_LEFT_LIGHT
VERTICAL_HEAVY_AND_RIGHT_LIGHT
VERTICAL_LIGHT_AND_HORIZONTAL_HEAVY
VERTICAL_LIGHT_AND_LEFT_HEAVY
VERTICAL_LIGHT_AND_RIGHT_HEAVY
VERTICAL_SINGLE_AND_HORIZONTAL_DOUBLE
VERTICAL_SINGLE_AND_LEFT_DOUBLE
VERTICAL_SINGLE_AND_RIGHT_DOUBLE

DOUBLE_DOWN_AND_HORIZONTAL = ''

DOUBLE_DOWN_AND_LEFT = ''

DOUBLE_DOWN_AND_RIGHT = ''

DOUBLE_HORIZONTAL = ''

DOUBLE_UP_AND_HORIZONTAL = ''

DOUBLE_UP_AND_LEFT = ''

DOUBLE_UP_AND_RIGHT = ''

DOUBLE_VERTICAL = ''

DOUBLE_VERTICAL_AND_HORIZONTAL = ''

DOUBLE_VERTICAL_AND_LEFT = ''

DOUBLE_VERTICAL_AND_RIGHT = ''

DOWN_DOUBLE_AND_HORIZONTAL_SINGLE = ''

DOWN_DOUBLE_AND_LEFT_SINGLE = ''

DOWN_DOUBLE_AND_RIGHT_SINGLE = ''

DOWN_HEAVY_AND_HORIZONTAL_LIGHT = ''

DOWN_HEAVY_AND_LEFT_LIGHT = ''

DOWN_HEAVY_AND_LEFT_UP_LIGHT = ''

DOWN_HEAVY_AND_RIGHT_LIGHT = ''

DOWN_HEAVY_AND_RIGHT_UP_LIGHT = ''

DOWN_HEAVY_AND_UP_HORIZONTAL_LIGHT = ''

DOWN_LIGHT_AND_HORIZONTAL_HEAVY = ''

DOWN_LIGHT_AND_LEFT_HEAVY = ''

DOWN_LIGHT_AND_LEFT_UP_HEAVY = ''

DOWN_LIGHT_AND_RIGHT_HEAVY = ''

DOWN_LIGHT_AND_RIGHT_UP_HEAVY = ''

DOWN_LIGHT_AND_UP_HORIZONTAL_HEAVY = ''

DOWN_SINGLE_AND_HORIZONTAL_DOUBLE = ''

DOWN_SINGLE_AND_LEFT_DOUBLE = ''

DOWN_SINGLE_AND_RIGHT_DOUBLE = ''

HEAVY_DOUBLE_DASH_HORIZONTAL = ''

HEAVY_DOUBLE_DASH_VERTICAL = ''

HEAVY_DOWN = ''

HEAVY_DOWN_AND_HORIZONTAL = ''

HEAVY_DOWN_AND_LEFT = ''

HEAVY_DOWN_AND_RIGHT = ''

HEAVY_HORIZONTAL = ''

HEAVY_LEFT = ''

HEAVY_LEFT_AND_LIGHT_RIGHT = ''

HEAVY_QUADRUPLE_DASH_HORIZONTAL = ''

HEAVY_QUADRUPLE_DASH_VERTICAL = ''

HEAVY_RIGHT = ''

HEAVY_TRIPLE_DASH_HORIZONTAL = ''

HEAVY_TRIPLE_DASH_VERTICAL = ''

HEAVY_UP = ''

HEAVY_UP_AND_HORIZONTAL = ''

HEAVY_UP_AND_LEFT = ''

HEAVY_UP_AND_LIGHT_DOWN = ''

HEAVY_UP_AND_RIGHT = ''

HEAVY_VERTICAL = ''

HEAVY_VERTICAL_AND_HORIZONTAL = ''

HEAVY_VERTICAL_AND_LEFT = ''

HEAVY_VERTICAL_AND_RIGHT = ''

LEFT_DOWN_HEAVY_AND_RIGHT_UP_LIGHT = ''

LEFT_HEAVY_AND_RIGHT_DOWN_LIGHT = ''

LEFT_HEAVY_AND_RIGHT_UP_LIGHT = ''

LEFT_HEAVY_AND_RIGHT_VERTICAL_LIGHT = ''

LEFT_LIGHT_AND_RIGHT_DOWN_HEAVY = ''

LEFT_LIGHT_AND_RIGHT_UP_HEAVY = ''

LEFT_LIGHT_AND_RIGHT_VERTICAL_HEAVY = ''

LEFT_UP_HEAVY_AND_RIGHT_DOWN_LIGHT = ''

LIGHT_ARC_DOWN_AND_LEFT = ''

LIGHT_ARC_DOWN_AND_RIGHT = ''

LIGHT_ARC_UP_AND_LEFT = ''

LIGHT_ARC_UP_AND_RIGHT = ''

LIGHT_DIAGONAL_CROSS = ''

LIGHT_DIAGONAL_UPPER_LEFT_TO_LOWER_RIGHT = '\'

LIGHT_DIAGONAL_UPPER_RIGHT_TO_LOWER_LEFT = ''

LIGHT_DOUBLE_DASH_HORIZONTAL = ''

LIGHT_DOUBLE_DASH_VERTICAL = ''

LIGHT_DOWN = ''

LIGHT_DOWN_AND_HORIZONTAL = ''

LIGHT_DOWN_AND_LEFT = ''

LIGHT_DOWN_AND_RIGHT = ''

LIGHT_HORIZONTAL = '─'

LIGHT_LEFT = ''

LIGHT_LEFT_AND_HEAVY_RIGHT = ''

LIGHT_QUADRUPLE_DASH_HORIZONTAL = ''

LIGHT_QUADRUPLE_DASH_VERTICAL = ''

LIGHT_RIGHT = ''

LIGHT_TRIPLE_DASH_HORIZONTAL = ''

LIGHT_TRIPLE_DASH_VERTICAL = ''

LIGHT_UP = ''

LIGHT_UP_AND_HEAVY_DOWN = ''

LIGHT_UP_AND_HORIZONTAL = ''

LIGHT_UP_AND_LEFT = ''

LIGHT_UP_AND_RIGHT = '└'

LIGHT_VERTICAL = '│'

LIGHT_VERTICAL_AND_HORIZONTAL = ''

LIGHT_VERTICAL_AND_LEFT = ''

LIGHT_VERTICAL_AND_RIGHT = '├'

RIGHT_DOWN_HEAVY_AND_LEFT_UP_LIGHT = ''

RIGHT_HEAVY_AND_LEFT_DOWN_LIGHT = ''

RIGHT_HEAVY_AND_LEFT_UP_LIGHT = ''

RIGHT_HEAVY_AND_LEFT_VERTICAL_LIGHT = ''

RIGHT_LIGHT_AND_LEFT_DOWN_HEAVY = ''

RIGHT_LIGHT_AND_LEFT_UP_HEAVY = ''

RIGHT_LIGHT_AND_LEFT_VERTICAL_HEAVY = ''

RIGHT_UP_HEAVY_AND_LEFT_DOWN_LIGHT = ''

UP_DOUBLE_AND_HORIZONTAL_SINGLE = ''

UP_DOUBLE_AND_LEFT_SINGLE = ''

UP_DOUBLE_AND_RIGHT_SINGLE = ''

UP_HEAVY_AND_DOWN_HORIZONTAL_LIGHT = ''

UP_HEAVY_AND_HORIZONTAL_LIGHT = ''

UP_HEAVY_AND_LEFT_DOWN_LIGHT = ''

UP_HEAVY_AND_LEFT_LIGHT = ''

UP_HEAVY_AND_RIGHT_DOWN_LIGHT = ''

UP_HEAVY_AND_RIGHT_LIGHT = ''

UP_LIGHT_AND_DOWN_HORIZONTAL_HEAVY = ''

UP_LIGHT_AND_HORIZONTAL_HEAVY = ''

UP_LIGHT_AND_LEFT_DOWN_HEAVY = ''

UP_LIGHT_AND_LEFT_HEAVY = ''

UP_LIGHT_AND_RIGHT_DOWN_HEAVY = ''

UP_LIGHT_AND_RIGHT_HEAVY = ''

UP_SINGLE_AND_HORIZONTAL_DOUBLE = ''

UP_SINGLE_AND_LEFT_DOUBLE = ''

UP_SINGLE_AND_RIGHT_DOUBLE = ''

VERTICAL_DOUBLE_AND_HORIZONTAL_SINGLE = ''

VERTICAL_DOUBLE_AND_LEFT_SINGLE = ''

VERTICAL_DOUBLE_AND_RIGHT_SINGLE = ''

VERTICAL_HEAVY_AND_HORIZONTAL_LIGHT = ''

VERTICAL_HEAVY_AND_LEFT_LIGHT = ''

VERTICAL_HEAVY_AND_RIGHT_LIGHT = ''

VERTICAL_LIGHT_AND_HORIZONTAL_HEAVY = ''

VERTICAL_LIGHT_AND_LEFT_HEAVY = ''

VERTICAL_LIGHT_AND_RIGHT_HEAVY = ''

VERTICAL_SINGLE_AND_HORIZONTAL_DOUBLE = ''

VERTICAL_SINGLE_AND_LEFT_DOUBLE = ''

VERTICAL_SINGLE_AND_RIGHT_DOUBLE = ''

GeometricShapes

class pygamelib.assets.graphics.**GeometricShapes**

Bases: object

Geometric shapes elements (unicode)

Here is the list of supported glyphs:

- BLACK_SQUARE =
- BLACK_LARGE_SQUARE =
- WHITE_SQUARE =
- WHITE_SQUARE_WITH_ROUNDED_CORNERS =
- WHITE_SQUARE_CONTAINING_BLACK_SMALL_SQUARE =
- SQUARE_WITH_HORIZONTAL_FILL =

- SQUARE_WITH_VERTICAL_FILL =
- SQUARE_WITH_ORTHOGONAL_CROSSHATCH_FILL =
- SQUARE_WITH_UPPER_LEFT_TO_LOWER_RIGHT_FILL =
- SQUARE_WITH_UPPER_RIGHT_TO_LOWER_LEFT_FILL =
- SQUARE_WITH_DIAGONAL_CROSSHATCH_FILL =
- BLACK_SMALL_SQUARE =
- WHITE_SMALL_SQUARE =
- BLACK_RECTANGLE =
- WHITE_RECTANGLE =
- BLACK_VERTICAL_RECTANGLE =
- WHITE_VERTICAL_RECTANGLE =
- BLACK_PARALLELOGRAM =
- WHITE_PARALLELOGRAM =
- BLACK_UP_POINTING_TRIANGLE =
- WHITE_UP_POINTING_TRIANGLE =
- BLACK_UP_POINTING_SMALL_TRIANGLE =
- WHITE_UP_POINTING_SMALL_TRIANGLE =
- BLACK_RIGHT_POINTING_TRIANGLE =
- WHITE_RIGHT_POINTING_TRIANGLE =
- BLACK_RIGHT_POINTING_SMALL_TRIANGLE =
- WHITE_RIGHT_POINTING_SMALL_TRIANGLE =
- BLACK_RIGHT_POINTING_POINTER =
- WHITE_RIGHT_POINTING_POINTER =
- BLACK_DOWN_POINTING_TRIANGLE =
- WHITE_DOWN_POINTING_TRIANGLE =
- BLACK_DOWN_POINTING_SMALL_TRIANGLE =
- WHITE_DOWN_POINTING_SMALL_TRIANGLE =
- BLACK_LEFT_POINTING_TRIANGLE =
- WHITE_LEFT_POINTING_TRIANGLE =
- BLACK_LEFT_POINTING_SMALL_TRIANGLE =
- WHITE_LEFT_POINTING_SMALL_TRIANGLE =
- BLACK_LEFT_POINTING_POINTER =
- WHITE_LEFT_POINTING_POINTER =
- BLACK_DIAMOND =
- WHITE_DIAMOND =
- WHITE_DIAMOND_CONTAINING_BLACK_SMALL_DIAMOND =

- FISHEYE =
- LOZENGE =
- WHITE_CIRCLE =
- DOTTED_CIRCLE =
- CIRCLE_WITH_VERTICAL_FILL =
- BULLSEYE =
- BLACK_CIRCLE =
- CIRCLE_WITH_LEFT_HALF_BLACK =
- CIRCLE_WITH_RIGHT_HALF_BLACK =
- CIRCLE_WITH_LOWER_HALF_BLACK =
- CIRCLE_WITH_UPPER_HALF_BLACK =
- CIRCLE_WITH_UPPER_RIGHT_QUADRANT_BLACK =
- CIRCLE_WITH_ALL_BUT_UPPER_LEFT_QUADRANT_BLACK =
- LEFT_HALF_BLACK_CIRCLE =
- RIGHT_HALF_BLACK_CIRCLE =
- INVERSE_BULLET =
- INVERSE_WHITE_CIRCLE =
- UPPER_HALF_INVERSE_WHITE_CIRCLE =
- LOWER_HALF_INVERSE_WHITE_CIRCLE =
- UPPER_LEFT_QUADRANT_CIRCULAR_ARC =
- UPPER_RIGHT_QUADRANT_CIRCULAR_ARC =
- LOWER_RIGHT_QUADRANT_CIRCULAR_ARC =
- LOWER_LEFT_QUADRANT_CIRCULAR_ARC =
- UPPER_HALF_CIRCLE =
- LOWER_HALF_CIRCLE =
- BLACK_LOWER_RIGHT_TRIANGLE =
- BLACK_LOWER_LEFT_TRIANGLE =
- BLACK_UPPER_LEFT_TRIANGLE =
- BLACK_UPPER_RIGHT_TRIANGLE =
- WHITE_BULLET = ◦
- BULLET = •
- RING_OPERATOR =
- SQUARE_WITH_LEFT_HALF_BLACK =
- SQUARE_WITH_RIGHT_HALF_BLACK =
- SQUARE_WITH_UPPER_LEFT_DIAGONAL_HALF_BLACK =
- SQUARE_WITH_LOWER_RIGHT_DIAGONAL_HALF_BLACK =

- WHITE_SQUARE_WITH_VERTICAL_BISECTING_LINE =
- WHITE_UP_POINTING_TRIANGLE_WITH_DOT =
- UP_POINTING_TRIANGLE_WITH_LEFT_HALF_BLACK =
- UP_POINTING_TRIANGLE_WITH_RIGHT_HALF_BLACK =
- LARGE_CIRCLE = ○
- WHITE_SQUARE_WITH_UPPER_LEFT_QUADRANT =
- WHITE_SQUARE_WITH_LOWER_LEFT_QUADRANT =
- WHITE_SQUARE_WITH_LOWER_RIGHT_QUADRANT =
- WHITE_SQUARE_WITH_UPPER_RIGHT_QUADRANT =
- WHITE_CIRCLE_WITH_UPPER_LEFT_QUADRANT =
- WHITE_CIRCLE_WITH_LOWER_LEFT_QUADRANT =
- WHITE_CIRCLE_WITH_LOWER_RIGHT_QUADRANT =
- WHITE_CIRCLE_WITH_UPPER_RIGHT_QUADRANT =
- UPPER_LEFT_TRIANGLE =
- UPPER_RIGHT_TRIANGLE =
- LOWER_LEFT_TRIANGLE =
- WHITE_MEDIUM_SQUARE =
- BLACK_MEDIUM_SQUARE =
- WHITE_MEDIUM_SMALL_SQUARE =
- BLACK_MEDIUM_SMALL_SQUARE =
- LOWER_RIGHT_TRIANGLE =

`__init__`()

Methods

`__init__`()

Attributes

BLACK_CIRCLE
BLACK_DIAMOND
BLACK_DOWN_POINTING_SMALL_TRIANGLE
BLACK_DOWN_POINTING_TRIANGLE

continues on next page

Table 3 – continued from previous page

BLACK_LARGE_SQUARE
BLACK_LEFT_POINTING_POINTER
BLACK_LEFT_POINTING_SMALL_TRIANGLE
BLACK_LEFT_POINTING_TRIANGLE
BLACK_LOWER_LEFT_TRIANGLE
BLACK_LOWER_RIGHT_TRIANGLE
BLACK_MEDIUM_SMALL_SQUARE
BLACK_MEDIUM_SQUARE
BLACK_PARALLELOGRAM
BLACK_RECTANGLE
BLACK_RIGHT_POINTING_POINTER
BLACK_RIGHT_POINTING_SMALL_TRIANGLE
BLACK_RIGHT_POINTING_TRIANGLE
BLACK_SMALL_SQUARE
BLACK_SQUARE
BLACK_UPPER_LEFT_TRIANGLE
BLACK_UPPER_RIGHT_TRIANGLE
BLACK_UP_POINTING_SMALL_TRIANGLE
BLACK_UP_POINTING_TRIANGLE
BLACK_VERTICAL_RECTANGLE
BULLET
BULLSEYE
CIRCLE_WITH_ALL_BUT_UPPER_LEFT_QUADRANT_B
CIRCLE_WITH_LEFT_HALF_BLACK
CIRCLE_WITH_LOWER_HALF_BLACK
CIRCLE_WITH_RIGHT_HALF_BLACK

continues on next page

Table 3 – continued from previous page

CIRCLE_WITH_UPPER_HALF_BLACK
CIRCLE_WITH_UPPER_RIGHT_QUADRANT_BLACK
CIRCLE_WITH_VERTICAL_FILL
DOTTED_CIRCLE
FISHEYE
INVERSE_BULLET
INVERSE_WHITE_CIRCLE
LARGE_CIRCLE
LEFT_HALF_BLACK_CIRCLE
LOWER_HALF_CIRCLE
LOWER_HALF_INVERSE_WHITE_CIRCLE
LOWER_LEFT_QUADRANT_CIRCULAR_ARC
LOWER_LEFT_TRIANGLE
LOWER_RIGHT_QUADRANT_CIRCULAR_ARC
LOWER_RIGHT_TRIANGLE
LOZENGE
RIGHT_HALF_BLACK_CIRCLE
RING_OPERATOR
SQUARE_WITH_DIAGONAL_CROSSHATCH_FILL
SQUARE_WITH_HORIZONTAL_FILL
SQUARE_WITH_LEFT_HALF_BLACK
SQUARE_WITH_LOWER_RIGHT_DIAGONAL_HALF_BLA
SQUARE_WITH_ORTHOGONAL_CROSSHATCH_FILL
SQUARE_WITH_RIGHT_HALF_BLACK
SQUARE_WITH_UPPER_LEFT_DIAGONAL_HALF_BLAC
SQUARE_WITH_UPPER_LEFT_TO_LOWER_RIGHT_FIL

continues on next page

Table 3 – continued from previous page

SQUARE_WITH_UPPER_RIGHT_TO_LOWER_LEFT_FIL
SQUARE_WITH_VERTICAL_FILL
UPPER_HALF_CIRCLE
UPPER_HALF_INVERSE_WHITE_CIRCLE
UPPER_LEFT_QUADRANT_CIRCULAR_ARC
UPPER_LEFT_TRIANGLE
UPPER_RIGHT_QUADRANT_CIRCULAR_ARC
UPPER_RIGHT_TRIANGLE
UP_POINTING_TRIANGLE_WITH_LEFT_HALF_BLACK
UP_POINTING_TRIANGLE_WITH_RIGHT_HALF_BLAC
WHITE_BULLET
WHITE_CIRCLE
WHITE_CIRCLE_WITH_LOWER_LEFT_QUADRANT
WHITE_CIRCLE_WITH_LOWER_RIGHT_QUADRANT
WHITE_CIRCLE_WITH_UPPER_LEFT_QUADRANT
WHITE_CIRCLE_WITH_UPPER_RIGHT_QUADRANT
WHITE_DIAMOND
WHITE_DIAMOND_CONTAINING_BLACK_SMALL_DIAM
WHITE_DOWN_POINTING_SMALL_TRIANGLE
WHITE_DOWN_POINTING_TRIANGLE
WHITE_LEFT_POINTING_POINTER
WHITE_LEFT_POINTING_SMALL_TRIANGLE
WHITE_LEFT_POINTING_TRIANGLE
WHITE_MEDIUM_SMALL_SQUARE
WHITE_MEDIUM_SQUARE
WHITE_PARALLELOGRAM

continues on next page

Table 3 – continued from previous page

WHITE_RECTANGLE
WHITE_RIGHT_POINTING_POINTER
WHITE_RIGHT_POINTING_SMALL_TRIANGLE
WHITE_RIGHT_POINTING_TRIANGLE
WHITE_SMALL_SQUARE
WHITE_SQUARE
WHITE_SQUARE_CONTAINING_BLACK_SMALL_SQUAR
WHITE_SQUARE_WITH_LOWER_LEFT_QUADRANT
WHITE_SQUARE_WITH_LOWER_RIGHT_QUADRANT
WHITE_SQUARE_WITH_ROUNDED_CORNERS
WHITE_SQUARE_WITH_UPPER_LEFT_QUADRANT
WHITE_SQUARE_WITH_UPPER_RIGHT_QUADRANT
WHITE_SQUARE_WITH_VERTICAL_BISECTING_LINE
WHITE_UP_POINTING_SMALL_TRIANGLE
WHITE_UP_POINTING_TRIANGLE
WHITE_UP_POINTING_TRIANGLE_WITH_DOT
WHITE_VERTICAL_RECTANGLE

BLACK_CIRCLE = ''

BLACK_DIAMOND = ''

BLACK_DOWN_POINTING_SMALL_TRIANGLE = ''

BLACK_DOWN_POINTING_TRIANGLE = ''

BLACK_LARGE_SQUARE = ''

BLACK_LEFT_POINTING_POINTER = ''

BLACK_LEFT_POINTING_SMALL_TRIANGLE = ''

BLACK_LEFT_POINTING_TRIANGLE = ''

BLACK_LOWER_LEFT_TRIANGLE = ''

BLACK_LOWER_RIGHT_TRIANGLE = ''

BLACK_MEDIUM_SMALL_SQUARE = ''

BLACK_MEDIUM_SQUARE = ''

BLACK_PARALLELOGRAM = ''

BLACK_RECTANGLE = ''

BLACK_RIGHT_POINTING_POINTER = ''

BLACK_RIGHT_POINTING_SMALL_TRIANGLE = ''

BLACK_RIGHT_POINTING_TRIANGLE = ''

BLACK_SMALL_SQUARE = ''

BLACK_SQUARE = ''

BLACK_UPPER_LEFT_TRIANGLE = ''

BLACK_UPPER_RIGHT_TRIANGLE = ''

BLACK_UP_POINTING_SMALL_TRIANGLE = ''

BLACK_UP_POINTING_TRIANGLE = ''

BLACK_VERTICAL_RECTANGLE = ''

BULLET = '•'

BULLSEYE = ''

CIRCLE_WITH_ALL_BUT_UPPER_LEFT_QUADRANT_BLACK = ''

CIRCLE_WITH_LEFT_HALF_BLACK = ''

CIRCLE_WITH_LOWER_HALF_BLACK = ''

CIRCLE_WITH_RIGHT_HALF_BLACK = ''

CIRCLE_WITH_UPPER_HALF_BLACK = ''

CIRCLE_WITH_UPPER_RIGHT_QUADRANT_BLACK = ''

CIRCLE_WITH_VERTICAL_FILL = ''

DOTTED_CIRCLE = ''

FISHEYE = ''

INVERSE_BULLET = ''

INVERSE_WHITE_CIRCLE = ''

LARGE_CIRCLE = '◯'

LEFT_HALF_BLACK_CIRCLE = ''

LOWER_HALF_CIRCLE = ''

LOWER_HALF_INVERSE_WHITE_CIRCLE = ''

LOWER_LEFT_QUADRANT_CIRCULAR_ARC = ''

LOWER_LEFT_TRIANGLE = ''

LOWER_RIGHT_QUADRANT_CIRCULAR_ARC = ''

LOWER_RIGHT_TRIANGLE = ''

LOZENGE = ''

RIGHT_HALF_BLACK_CIRCLE = ''

RING_OPERATOR = ''

SQUARE_WITH_DIAGONAL_CROSSHATCH_FILL = ''

SQUARE_WITH_HORIZONTAL_FILL = ''

SQUARE_WITH_LEFT_HALF_BLACK = ''

SQUARE_WITH_LOWER_RIGHT_DIAGONAL_HALF_BLACK = ''

SQUARE_WITH_ORTHOGONAL_CROSSHATCH_FILL = ''

SQUARE_WITH_RIGHT_HALF_BLACK = ''

SQUARE_WITH_UPPER_LEFT_DIAGONAL_HALF_BLACK = ''

SQUARE_WITH_UPPER_LEFT_TO_LOWER_RIGHT_FILL = ''

SQUARE_WITH_UPPER_RIGHT_TO_LOWER_LEFT_FILL = ''

SQUARE_WITH_VERTICAL_FILL = ''

UPPER_HALF_CIRCLE = ''

UPPER_HALF_INVERSE_WHITE_CIRCLE = ''

UPPER_LEFT_QUADRANT_CIRCULAR_ARC = ''

UPPER_LEFT_TRIANGLE = ''

UPPER_RIGHT_QUADRANT_CIRCULAR_ARC = ''

UPPER_RIGHT_TRIANGLE = ''

UP_POINTING_TRIANGLE_WITH_LEFT_HALF_BLACK = ''

UP_POINTING_TRIANGLE_WITH_RIGHT_HALF_BLACK = ''

WHITE_BULLET = '◦'

WHITE_CIRCLE = ''

WHITE_CIRCLE_WITH_LOWER_LEFT_QUADRANT = ''

WHITE_CIRCLE_WITH_LOWER_RIGHT_QUADRANT = ''

WHITE_CIRCLE_WITH_UPPER_LEFT_QUADRANT = ''

WHITE_CIRCLE_WITH_UPPER_RIGHT_QUADRANT = ''

WHITE_DIAMOND = ''

WHITE_DIAMOND_CONTAINING_BLACK_SMALL_DIAMOND = ''

WHITE_DOWN_POINTING_SMALL_TRIANGLE = ''

WHITE_DOWN_POINTING_TRIANGLE = ''

WHITE_LEFT_POINTING_POINTER = ''

WHITE_LEFT_POINTING_SMALL_TRIANGLE = ''

WHITE_LEFT_POINTING_TRIANGLE = ''

WHITE_MEDIUM_SMALL_SQUARE = ''

WHITE_MEDIUM_SQUARE = ''

WHITE_PARALLELOGRAM = ''

WHITE_RECTANGLE = ''

WHITE_RIGHT_POINTING_POINTER = ''

WHITE_RIGHT_POINTING_SMALL_TRIANGLE = ''

WHITE_RIGHT_POINTING_TRIANGLE = ''

WHITE_SMALL_SQUARE = ''

WHITE_SQUARE = ''

WHITE_SQUARE_CONTAINING_BLACK_SMALL_SQUARE = ''

WHITE_SQUARE_WITH_LOWER_LEFT_QUADRANT = ''

WHITE_SQUARE_WITH_LOWER_RIGHT_QUADRANT = ''

WHITE_SQUARE_WITH_ROUNDED_CORNERS = ''

WHITE_SQUARE_WITH_UPPER_LEFT_QUADRANT = ''

WHITE_SQUARE_WITH_UPPER_RIGHT_QUADRANT = ''

WHITE_SQUARE_WITH_VERTICAL_BISECTING_LINE = ''

WHITE_UP_POINTING_SMALL_TRIANGLE = ''

WHITE_UP_POINTING_TRIANGLE = ''

WHITE_UP_POINTING_TRIANGLE_WITH_DOT = ''

WHITE_VERTICAL_RECTANGLE = ''

MiscTechnicals

class pygamelib.assets.graphics.**MiscTechnicals**

Bases: `object`

Miscellanous Technical block (unicode)

Here is the list of supported glyphs:

- DIAMETER_SIGN = “”
- ELECTRIC_ARROW = “”
- HOUSE = “”
- UP_ARROWHEAD = “”
- DOWN_ARROWHEAD = “”
- PROJECTIVE = “”
- PERSPECTIVE = “”
- WAVY_LINE = “”
- LEFT_CEILING = “”
- RIGHT_CEILING = “”
- LEFT_FLOOR = “”
- RIGHT_FLOOR = “”
- BOTTOM_RIGHT_CROP = “”
- BOTTOM_LEFT_CROP = “”
- TOP_RIGHT_CROP = “”
- TOP_LEFT_CROP = “”
- REVERSED_NOT_SIGN = “”
- SQUARE_LOZENGE = “”
- ARC = “”
- SEGMENT = “”
- SECTOR = “”
- TELEPHONE_RECORDER = “”
- POSITION_INDICATOR = “”
- VIEWDATA_SQUARE = “”
- PLACE_OF_INTEREST_SIGN = “”
- TURNED_NOT_SIGN = “”
- WATCH = “”
- HOURGLASS = “”
- TOP_LEFT_CORNER = “”
- TOP_RIGHT_CORNER = “”
- BOTTOM_LEFT_CORNER = “”

- BOTTOM_RIGHT_CORNER = “”
- TOP_HALF_INTEGRAL = “”
- BOTTOM_HALF_INTEGRAL = “”
- FROWN = “”
- SMILE = “”
- UP_ARROWHEAD_BETWEEN_TWO_HORIZONTAL_BARS = “”
- OPTION_KEY = “”
- ERASE_TO_THE_RIGHT = “”
- X_IN_A_RECTANGLE_BOX = “”
- KEYBOARD = “”
- LEFT_POINTING_ANGLE_BRACKET = “〈”
- RIGHT_POINTING_ANGLE_BRACKET = “〉”
- ERASE_TO_THE_LEFT = “”
- BENZENE_RING = “”
- CYLINDRICITY = “”
- ALL_AROUND_PROFILE = “”
- SYMMETRY = “”
- TOTAL_RUNOUT = “”
- DIMENSION_ORIGIN = “”
- CONICAL_TAPER = “”
- SLOPE = “”
- COUNTERBORE = “”
- COUNTERSINK = “”
- APL_FUNCTIONAL_SYMBOL_I_BEAM = “”
- APL_FUNCTIONAL_SYMBOL_SQUISH_QUAD = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_EQUAL = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_DIVIDE = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_DIAMOND = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_JOT = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_CIRCLE = “”
- APL_FUNCTIONAL_SYMBOL_CIRCLE_STILE = “”
- APL_FUNCTIONAL_SYMBOL_CIRCLE_JOT = “”
- APL_FUNCTIONAL_SYMBOL_SLASH_BAR = “”
- APL_FUNCTIONAL_SYMBOL_BACKSLASH_BAR = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_SLASH = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_BACKSLASH = “”

- APL_FUNCTIONAL_SYMBOL_QUAD_LESS_THAN = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_GREATER_THAN = “”
- APL_FUNCTIONAL_SYMBOL_LEFTWARDS_VANE = “”
- APL_FUNCTIONAL_SYMBOL_RIGHTWARDS_VANE = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_LEFTWARDS_ARROW = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_RIGHTWARDS_ARROW = “”
- APL_FUNCTIONAL_SYMBOL_CIRCLE_BACKSLASH = “”
- APL_FUNCTIONAL_SYMBOL_DOWN_TACK_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_DELTA_STILE = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_DOWN_CARET = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_DELTA = “”
- APL_FUNCTIONAL_SYMBOL_DOWN_TACK_JOT = “”
- APL_FUNCTIONAL_SYMBOL_UPWARDS_VANE = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_UPWARDS_ARROW = “”
- APL_FUNCTIONAL_SYMBOL_UP_TACK_OVERBAR = “”
- APL_FUNCTIONAL_SYMBOL_DEL_STILE = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_UP_CARET = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_DEL = “”
- APL_FUNCTIONAL_SYMBOL_UP_TACK_JOT = “”
- APL_FUNCTIONAL_SYMBOL_DOWNWARDS_VANE = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_DOWNWARDS_ARROW = “”
- APL_FUNCTIONAL_SYMBOL_QUOTE_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_DELTA_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_DIAMOND_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_JOT_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_CIRCLE_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_UP_SHOE_JOT = “”
- APL_FUNCTIONAL_SYMBOL_QUOTE_QUAD = “”
- APL_FUNCTIONAL_SYMBOL_CIRCLE_STAR = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_COLON = “”
- APL_FUNCTIONAL_SYMBOL_UP_TACK_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_DEL_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_STAR_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_JOT_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_CIRCLE_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_DOWN_SHOE_STILE = “”

- APL_FUNCTIONAL_SYMBOL_LEFT_SHOE_STILE = “”
- APL_FUNCTIONAL_SYMBOL_TILDE_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_GREATER_THAN_DIAERESIS = “”
- APL_FUNCTIONAL_SYMBOL_COMMA_BAR = “”
- APL_FUNCTIONAL_SYMBOL_DEL_TILDE = “”
- APL_FUNCTIONAL_SYMBOL_ZILDE = “”
- APL_FUNCTIONAL_SYMBOL_STILE_TILDE = “”
- APL_FUNCTIONAL_SYMBOL_SEMICOLON_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_NOT_EQUAL = “”
- APL_FUNCTIONAL_SYMBOL_QUAD_QUESTION = “”
- APL_FUNCTIONAL_SYMBOL_DOWN_CARET_TILDE = “”
- APL_FUNCTIONAL_SYMBOL_UP_CARET_TILDE = “”
- APL_FUNCTIONAL_SYMBOL_IOTA = “”
- APL_FUNCTIONAL_SYMBOL_RHO = “”
- APL_FUNCTIONAL_SYMBOL_OMEGA = “”
- APL_FUNCTIONAL_SYMBOL_ALPHA_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_EPSILON_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_IOTA_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_OMEGA_UNDERBAR = “”
- APL_FUNCTIONAL_SYMBOL_ALPHA = “”
- NOT_CHECK_MARK = “”
- RIGHT_ANGLE_WITH_DOWNWARDS_ZIGZAG_ARROW = “”
- SHOULDERED_OPEN_BOX = “”
- BELL_SYMBOL = “”
- VERTICAL_LINE_WITH_MIDDLE_DOT = “”
- INSERTION_SYMBOL = “”
- CONTINUOUS_UNDERLINE_SYMBOL = “”
- DISCONTINUOUS_UNDERLINE_SYMBOL = “”
- EMPHASIS_SYMBOL = “”
- COMPOSITION_SYMBOL = “”
- WHITE_SQUARE_WITH_CENTRE_VERTICAL_LINE = “”
- ENTER_SYMBOL = “”
- ALTERNATIVE_KEY_SYMBOL = “”
- HELM_SYMBOL = “”
- CIRCLED_HORIZONTAL_BAR_WITH_NOTCH = “”
- CIRCLED_TRIANGLE_DOWN = “”

- BROKEN_CIRCLE_WITH_NORTHWEST_ARROW = “”
- UNDO_SYMBOL = “”
- MONOSTABLE_SYMBOL = “”
- HYSTERESIS_SYMBOL = “”
- OPEN_CIRCUIT_OUTPUT_H_TYPE_SYMBOL = “”
- OPEN_CIRCUIT_OUTPUT_L_TYPE_SYMBOL = “”
- PASSIVE_PULL_DOWN_OUTPUT_SYMBOL = “”
- PASSIVE_PULL_UP_OUTPUT_SYMBOL = “”
- DIRECT_CURRENT_SYMBOL_FORM_TWO = “”
- SOFTWARE_FUNCTION_SYMBOL = “”
- APL_FUNCTIONAL_SYMBOL_QUAD = “”
- DECIMAL_SEPARATOR_KEY_SYMBOL = “”
- PREVIOUS_PAGE = “”
- NEXT_PAGE = “”
- PRINT_SCREEN_SYMBOL = “”
- CLEAR_SCREEN_SYMBOL = “”
- LEFT_PARENTHESIS_UPPER_HOOK = “”
- LEFT_PARENTHESIS_EXTENSION = “”
- LEFT_PARENTHESIS_LOWER_HOOK = “”
- RIGHT_PARENTHESIS_UPPER_HOOK = “”
- RIGHT_PARENTHESIS_EXTENSION = “”
- RIGHT_PARENTHESIS_LOWER_HOOK = “”
- LEFT_SQUARE_BRACKET_UPPER_CORNER = “”
- LEFT_SQUARE_BRACKET_EXTENSION = “”
- LEFT_SQUARE_BRACKET_LOWER_CORNER = “”
- RIGHT_SQUARE_BRACKET_UPPER_CORNER = “”
- RIGHT_SQUARE_BRACKET_EXTENSION = “”
- RIGHT_SQUARE_BRACKET_LOWER_CORNER = “”
- LEFT_CURLY_BRACKET_UPPER_HOOK = “”
- LEFT_CURLY_BRACKET_MIDDLE_PIECE = “”
- LEFT_CURLY_BRACKET_LOWER_HOOK = “”
- CURLY_BRACKET_EXTENSION = “”
- RIGHT_CURLY_BRACKET_UPPER_HOOK = “”
- RIGHT_CURLY_BRACKET_MIDDLE_PIECE = “”
- RIGHT_CURLY_BRACKET_LOWER_HOOK = “”
- INTEGRAL_EXTENSION = “”

- HORIZONTAL_LINE_EXTENSION = “”
- UPPER_LEFT_OR_LOWER_RIGHT_CURLY_BRACKET_SECTION = “”
- UPPER_RIGHT_OR_LOWER_LEFT_CURLY_BRACKET_SECTION = “”
- SUMMATION_TOP = “”
- SUMMATION_BOTTOM = “”
- TOP_SQUARE_BRACKET = “”
- BOTTOM_SQUARE_BRACKET = “”
- BOTTOM_SQUARE_BRACKET_OVER_TOP_SQUARE_BRACKET = “”
- RADICAL_SYMBOL_BOTTOM = “”
- LEFT_VERTICAL_BOX_LINE = “”
- RIGHT_VERTICAL_BOX_LINE = “”
- HORIZONTAL_SCAN_LINE_1 = “”
- HORIZONTAL_SCAN_LINE_3 = “”
- HORIZONTAL_SCAN_LINE_7 = “”
- HORIZONTAL_SCAN_LINE_9 = “_”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_TOP_RIGHT = “”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_BOTTOM_RIGHT = “”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_WITH_CIRCLE = “”
- DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL_WITH_CIRCLE = “”
- DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_WITH_CIRCLE = “”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_WITH_TRIANGLE = “”
- DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL_WITH_TRIANGLE = “”
- DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_WITH_TRIANGLE = “”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_WAVE = “”
- DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL_WITH_WAVE = “”
- DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_WITH_WAVE = “”
- DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL = “”
- DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL = “”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_TOP_LEFT = “”
- DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_BOTTOM_LEFT = “”
- SQUARE_FOOT = “”
- RETURN_SYMBOL = “”
- EJECT_SYMBOL = “”
- VERTICAL_LINE_EXTENSION = “”
- METRICAL_BREVE = “”
- METRICAL_LONG_OVER_SHORT = “”

- METRICAL_SHORT_OVER_LONG = “”
- METRICAL_LONG_OVER_TWO_SHORTS = “”
- METRICAL_TWO_SHORTS_OVER_LONG = “”
- METRICAL_TWO_SHORTS_JOINED = “”
- METRICAL_TRISEME = “”
- METRICAL_TETRASEME = “”
- METRICAL_PENTASEME = “”
- EARTH_GROUND = “”
- FUSE = “”
- TOP_PARENTHESIS = “”
- BOTTOM_PARENTHESIS = “”
- TOP_CURLY_BRACKET = “”
- BOTTOM_CURLY_BRACKET = “”
- TOP_TORTOISE_SHELL_BRACKET = “”
- BOTTOM_TORTOISE_SHELL_BRACKET = “”
- WHITE_TRAPEZIUM = “”
- BENZENE_RING_WITH_CIRCLE = “”
- STRAIGHTNESS = “”
- FLATNESS = “”
- AC_CURRENT = “”
- ELECTRICAL_INTERSECTION = “”
- DECIMAL_EXPONENT_SYMBOL = “”
- BLACK_RIGHT_POINTING_DOUBLE_TRIANGLE = “”
- BLACK_LEFT_POINTING_DOUBLE_TRIANGLE = “”
- BLACK_UP_POINTING_DOUBLE_TRIANGLE = “”
- BLACK_DOWN_POINTING_DOUBLE_TRIANGLE = “”
- BLACK_RIGHT_POINTING_DOUBLE_TRIANGLE_WITH_VERTICAL_BAR = “”
- BLACK_LEFT_POINTING_DOUBLE_TRIANGLE_WITH_VERTICAL_BAR = “”
- BLACK_RIGHT_POINTING_TRIANGLE_WITH_DOUBLE_VERTICAL_BAR = “”
- ALARM_CLOCK = “”
- STOPWATCH = “”
- TIMER_CLOCK = “”
- HOURGLASS_WITH_FLOWING_SAND = “”
- BLACK_MEDIUM_LEFT_POINTING_TRIANGLE = “”
- BLACK_MEDIUM_RIGHT_POINTING_TRIANGLE = “”
- BLACK_MEDIUM_UP_POINTING_TRIANGLE = “”

- BLACK_MEDIUM_DOWN_POINTING_TRIANGLE = “”
- DOUBLE_VERTICAL_BAR = “”
- BLACK_SQUARE_FOR_STOP = “”
- BLACK_CIRCLE_FOR_RECORD = “”
- POWER_SYMBOL = “”
- POWER_ON_OFF_SYMBOL = “”
- POWER_ON_SYMBOL = “”
- POWER_SLEEP_SYMBOL = “”
- OBSERVER_EYE_SYMBOL = “”

Attributes

DIAMETER_SIGN
ELECTRIC_ARROW
HOUSE
UP_ARROWHEAD
DOWN_ARROWHEAD
PROJECTIVE
PERSPECTIVE
WAVY_LINE
LEFT_CEILING
RIGHT_CEILING
LEFT_FLOOR
RIGHT_FLOOR
BOTTOM_RIGHT_CROP
BOTTOM_LEFT_CROP
TOP_RIGHT_CROP
TOP_LEFT_CROP
REVERSED_NOT_SIGN

continues on next page

Table 4 – continued from previous page

SQUARE_LOZENGE
ARC
SEGMENT
SECTOR
TELEPHONE_RECORDER
POSITION_INDICATOR
VIEWDATA_SQUARE
PLACE_OF_INTEREST_SIGN
TURNED_NOT_SIGN
WATCH
HOURGLASS
TOP_LEFT_CORNER
TOP_RIGHT_CORNER
BOTTOM_LEFT_CORNER
BOTTOM_RIGHT_CORNER
TOP_HALF_INTEGRAL
BOTTOM_HALF_INTEGRAL
FROWN
SMILE
UP_ARROWHEAD_BETWEEN_TWO_HORIZONTAL_BARS
OPTION_KEY
ERASE_TO_THE_RIGHT
X_IN_A_RECTANGLE_BOX
KEYBOARD
LEFT_POINTING_ANGLE_BRACKET
RIGHT_POINTING_ANGLE_BRACKET

continues on next page

Table 4 – continued from previous page

ERASE_TO_THE_LEFT
BENZENE_RING
CYLINDRICITY
ALL_AROUND_PROFILE
SYMMETRY
TOTAL_RUNOUT
DIMENSION_ORIGIN
CONICAL_TAPER
SLOPE
COUNTERBORE
COUNTERSINK
APL_FUNCTIONAL_SYMBOL_I_BEAM
APL_FUNCTIONAL_SYMBOL_SQUISH_QUAD
APL_FUNCTIONAL_SYMBOL_QUAD_EQUAL
APL_FUNCTIONAL_SYMBOL_QUAD_DIVIDE
APL_FUNCTIONAL_SYMBOL_QUAD_DIAMOND
APL_FUNCTIONAL_SYMBOL_QUAD_JOT
APL_FUNCTIONAL_SYMBOL_QUAD_CIRCLE
APL_FUNCTIONAL_SYMBOL_CIRCLE_STILE
APL_FUNCTIONAL_SYMBOL_CIRCLE_JOT
APL_FUNCTIONAL_SYMBOL_SLASH_BAR
APL_FUNCTIONAL_SYMBOL_BACKSLASH_BAR
APL_FUNCTIONAL_SYMBOL_QUAD_SLASH
APL_FUNCTIONAL_SYMBOL_QUAD_BACKSLASH
APL_FUNCTIONAL_SYMBOL_QUAD_LESS_THAN
APL_FUNCTIONAL_SYMBOL_QUAD_GREATER_THAN

continues on next page

Table 4 – continued from previous page

APL_FUNCTIONAL_SYMBOL_LEFTWARDS_VANE
APL_FUNCTIONAL_SYMBOL_RIGHTWARDS_VANE
APL_FUNCTIONAL_SYMBOL_QUAD_LEFTWARDS_ARRO
APL_FUNCTIONAL_SYMBOL_QUAD_RIGHTWARDS_ARR
APL_FUNCTIONAL_SYMBOL_CIRCLE_BACKSLASH
APL_FUNCTIONAL_SYMBOL_DOWN_TACK_UNDERBAR
APL_FUNCTIONAL_SYMBOL_DELTA_STILE
APL_FUNCTIONAL_SYMBOL_QUAD_DOWN_CARET
APL_FUNCTIONAL_SYMBOL_QUAD_DELTA
APL_FUNCTIONAL_SYMBOL_DOWN_TACK_JOT
APL_FUNCTIONAL_SYMBOL_UPWARDS_VANE
APL_FUNCTIONAL_SYMBOL_QUAD_UPWARDS_ARROW
APL_FUNCTIONAL_SYMBOL_UP_TACK_OVERBAR
APL_FUNCTIONAL_SYMBOL_DEL_STILE
APL_FUNCTIONAL_SYMBOL_QUAD_UP_CARET
APL_FUNCTIONAL_SYMBOL_QUAD_DEL
APL_FUNCTIONAL_SYMBOL_UP_TACK_JOT
APL_FUNCTIONAL_SYMBOL_DOWNWARDS_VANE
APL_FUNCTIONAL_SYMBOL_QUAD_DOWNWARDS_ARRO
APL_FUNCTIONAL_SYMBOL_QUOTE_UNDERBAR
APL_FUNCTIONAL_SYMBOL_DELTA_UNDERBAR
APL_FUNCTIONAL_SYMBOL_DIAMOND_UNDERBAR
APL_FUNCTIONAL_SYMBOL_JOT_UNDERBAR
APL_FUNCTIONAL_SYMBOL_CIRCLE_UNDERBAR
APL_FUNCTIONAL_SYMBOL_UP_SHOE_JOT
APL_FUNCTIONAL_SYMBOL_QUOTE_QUAD

continues on next page

Table 4 – continued from previous page

APL_FUNCTIONAL_SYMBOL_CIRCLE_STAR
APL_FUNCTIONAL_SYMBOL_QUAD_COLON
APL_FUNCTIONAL_SYMBOL_UP_TACK_DIAERESIS
APL_FUNCTIONAL_SYMBOL_DEL_DIAERESIS
APL_FUNCTIONAL_SYMBOL_STAR_DIAERESIS
APL_FUNCTIONAL_SYMBOL_JOT_DIAERESIS
APL_FUNCTIONAL_SYMBOL_CIRCLE_DIAERESIS
APL_FUNCTIONAL_SYMBOL_DOWN_SHOE_STILE
APL_FUNCTIONAL_SYMBOL_LEFT_SHOE_STILE
APL_FUNCTIONAL_SYMBOL_TILDE_DIAERESIS
APL_FUNCTIONAL_SYMBOL_GREATER_THAN_DIAERE
APL_FUNCTIONAL_SYMBOL_COMMA_BAR
APL_FUNCTIONAL_SYMBOL_DEL_TILDE
APL_FUNCTIONAL_SYMBOL_ZILDE
APL_FUNCTIONAL_SYMBOL_STILE_TILDE
APL_FUNCTIONAL_SYMBOL_SEMICOLON_UNDERBAR
APL_FUNCTIONAL_SYMBOL_QUAD_NOT_EQUAL
APL_FUNCTIONAL_SYMBOL_QUAD_QUESTION
APL_FUNCTIONAL_SYMBOL_DOWN_CARET_TILDE
APL_FUNCTIONAL_SYMBOL_UP_CARET_TILDE
APL_FUNCTIONAL_SYMBOL_IOTA
APL_FUNCTIONAL_SYMBOL_RHO
APL_FUNCTIONAL_SYMBOL_OMEGA
APL_FUNCTIONAL_SYMBOL_ALPHA_UNDERBAR
APL_FUNCTIONAL_SYMBOL_EPSILON_UNDERBAR
APL_FUNCTIONAL_SYMBOL_IOTA_UNDERBAR

continues on next page

Table 4 – continued from previous page

APL_FUNCTIONAL_SYMBOL_OMEGA_UNDERBAR
APL_FUNCTIONAL_SYMBOL_ALPHA
NOT_CHECK_MARK
RIGHT_ANGLE_WITH_DOWNWARDS_ZIGZAG_ARROW
SHOULDERED_OPEN_BOX
BELL_SYMBOL
VERTICAL_LINE_WITH_MIDDLE_DOT
INSERTION_SYMBOL
CONTINUOUS_UNDERLINE_SYMBOL
DISCONTINUOUS_UNDERLINE_SYMBOL
EMPHASIS_SYMBOL
COMPOSITION_SYMBOL
WHITE_SQUARE_WITH_CENTRE_VERTICAL_LINE
ENTER_SYMBOL
ALTERNATIVE_KEY_SYMBOL
HELM_SYMBOL
CIRCLED_HORIZONTAL_BAR_WITH_NOTCH
CIRCLED_TRIANGLE_DOWN
BROKEN_CIRCLE_WITH_NORTHWEST_ARROW
UNDO_SYMBOL
MONOSTABLE_SYMBOL
HYSTERESIS_SYMBOL
OPEN_CIRCUIT_OUTPUT_H_TYPE_SYMBOL
OPEN_CIRCUIT_OUTPUT_L_TYPE_SYMBOL
PASSIVE_PULL_DOWN_OUTPUT_SYMBOL
PASSIVE_PULL_UP_OUTPUT_SYMBOL

continues on next page

Table 4 – continued from previous page

DIRECT_CURRENT_SYMBOL_FORM_TWO
SOFTWARE_FUNCTION_SYMBOL
APL_FUNCTIONAL_SYMBOL_QUAD
DECIMAL_SEPARATOR_KEY_SYMBOL
PREVIOUS_PAGE
NEXT_PAGE
PRINT_SCREEN_SYMBOL
CLEAR_SCREEN_SYMBOL
LEFT_PARENTHESIS_UPPER_HOOK
LEFT_PARENTHESIS_EXTENSION
LEFT_PARENTHESIS_LOWER_HOOK
RIGHT_PARENTHESIS_UPPER_HOOK
RIGHT_PARENTHESIS_EXTENSION
RIGHT_PARENTHESIS_LOWER_HOOK
LEFT_SQUARE_BRACKET_UPPER_CORNER
LEFT_SQUARE_BRACKET_EXTENSION
LEFT_SQUARE_BRACKET_LOWER_CORNER
RIGHT_SQUARE_BRACKET_UPPER_CORNER
RIGHT_SQUARE_BRACKET_EXTENSION
RIGHT_SQUARE_BRACKET_LOWER_CORNER
LEFT_CURLY_BRACKET_UPPER_HOOK
LEFT_CURLY_BRACKET_MIDDLE_PIECE
LEFT_CURLY_BRACKET_LOWER_HOOK
CURLY_BRACKET_EXTENSION
RIGHT_CURLY_BRACKET_UPPER_HOOK
RIGHT_CURLY_BRACKET_MIDDLE_PIECE

continues on next page

Table 4 – continued from previous page

RIGHT_CURLY_BRACKET_LOWER_HOOK
INTEGRAL_EXTENSION
HORIZONTAL_LINE_EXTENSION
UPPER_LEFT_OR_LOWER_RIGHT_CURLY_BRACKET_S
UPPER_RIGHT_OR_LOWER_LEFT_CURLY_BRACKET_S
SUMMATION_TOP
SUMMATION_BOTTOM
TOP_SQUARE_BRACKET
BOTTOM_SQUARE_BRACKET
BOTTOM_SQUARE_BRACKET_OVER_TOP_SQUARE_BRA
RADICAL_SYMBOL_BOTTOM
LEFT_VERTICAL_BOX_LINE
RIGHT_VERTICAL_BOX_LINE
HORIZONTAL_SCAN_LINE_1
HORIZONTAL_SCAN_LINE_3
HORIZONTAL_SCAN_LINE_7
HORIZONTAL_SCAN_LINE_9
DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_TOP_R
DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_BOTTO
DENTISTRY_SYMBOL_LIGHT_VERTICAL_WITH_CIRC
DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTA
*DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_*
DENTISTRY_SYMBOL_LIGHT_VERTICAL_WITH_TRIA
DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTA
*DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_*
DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_WAVE

continues on next page

Table 4 – continued from previous page

DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTA
*DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_*
DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTA
DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL
DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_TOP_L
DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_BOTTO
SQUARE_FOOT
RETURN_SYMBOL
EJECT_SYMBOL
VERTICAL_LINE_EXTENSION
METRICAL_BREVE
METRICAL_LONG_OVER_SHORT
METRICAL_SHORT_OVER_LONG
METRICAL_LONG_OVER_TWO_SHORTS
METRICAL_TWO_SHORTS_OVER_LONG
METRICAL_TWO_SHORTS_JOINED
METRICAL_TRISEME
METRICAL_TETRASEME
METRICAL_PENTASEME
EARTH_GROUND
FUSE
TOP_PARENTHESIS
BOTTOM_PARENTHESIS
TOP_CURLY_BRACKET
BOTTOM_CURLY_BRACKET
TOP_TORTOISE_SHELL_BRACKET

continues on next page

Table 4 – continued from previous page

BOTTOM_TORTOISE_SHELL_BRACKET
WHITE_TRAPEZIUM
BENZENE_RING_WITH_CIRCLE
STRAIGHTNESS
FLATNESS
AC_CURRENT
ELECTRICAL_INTERSECTION
DECIMAL_EXPONENT_SYMBOL
BLACK_RIGHT_POINTING_DOUBLE_TRIANGLE
BLACK_LEFT_POINTING_DOUBLE_TRIANGLE
BLACK_UP_POINTING_DOUBLE_TRIANGLE
BLACK_DOWN_POINTING_DOUBLE_TRIANGLE
BLACK_RIGHT_POINTING_DOUBLE_TRIANGLE_WITH
*BLACK_LEFT_POINTING_DOUBLE_TRIANGLE_WITH_*
BLACK_RIGHT_POINTING_TRIANGLE_WITH_DOUBLE
ALARM_CLOCK
STOPWATCH
TIMER_CLOCK
HOURGLASS_WITH_FLOWING_SAND
BLACK_MEDIUM_LEFT_POINTING_TRIANGLE
BLACK_MEDIUM_RIGHT_POINTING_TRIANGLE
BLACK_MEDIUM_UP_POINTING_TRIANGLE
BLACK_MEDIUM_DOWN_POINTING_TRIANGLE
DOUBLE_VERTICAL_BAR
BLACK_SQUARE_FOR_STOP
BLACK_CIRCLE_FOR_RECORD

continues on next page

Table 4 – continued from previous page

POWER_SYMBOL
POWER_ON_OFF_SYMBOL
POWER_ON_SYMBOL
POWER_SLEEP_SYMBOL
OBSERVER_EYE_SYMBOL

AC_CURRENT = ''

ALARM_CLOCK = ''

ALL_AROUND_PROFILE = ''

ALTERNATIVE_KEY_SYMBOL = ''

APL_FUNCTIONAL_SYMBOL_ALPHA = ''

APL_FUNCTIONAL_SYMBOL_ALPHA_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_BACKSLASH_BAR = ''

APL_FUNCTIONAL_SYMBOL_CIRCLE_BACKSLASH = ''

APL_FUNCTIONAL_SYMBOL_CIRCLE_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_CIRCLE_JOT = ''

APL_FUNCTIONAL_SYMBOL_CIRCLE_STAR = ''

APL_FUNCTIONAL_SYMBOL_CIRCLE_STILE = ''

APL_FUNCTIONAL_SYMBOL_CIRCLE_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_COMMA_BAR = ''

APL_FUNCTIONAL_SYMBOL_DELTA_STILE = ''

APL_FUNCTIONAL_SYMBOL_DELTA_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_DEL_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_DEL_STILE = ''

APL_FUNCTIONAL_SYMBOL_DEL_TILDE = ''

APL_FUNCTIONAL_SYMBOL_DIAMOND_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_DOWNWARDS_VANE = ''

APL_FUNCTIONAL_SYMBOL_DOWN_CARET_TILDE = ''

APL_FUNCTIONAL_SYMBOL_DOWN_SHOE_STILE = ''

APL_FUNCTIONAL_SYMBOL_DOWN_TACK_JOT = ''

APL_FUNCTIONAL_SYMBOL_DOWN_TACK_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_EPSILON_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_GREATER_THAN_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_IOTA = ''

APL_FUNCTIONAL_SYMBOL_IOTA_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_I_BEAM = ''

APL_FUNCTIONAL_SYMBOL_JOT_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_JOT_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_LEFTWARDS_VANE = ''

APL_FUNCTIONAL_SYMBOL_LEFT_SHOE_STILE = ''

APL_FUNCTIONAL_SYMBOL_OMEGA = ''

APL_FUNCTIONAL_SYMBOL_OMEGA_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_QUAD = ''

APL_FUNCTIONAL_SYMBOL_QUAD_BACKSLASH = ''

APL_FUNCTIONAL_SYMBOL_QUAD_CIRCLE = ''

APL_FUNCTIONAL_SYMBOL_QUAD_COLON = ''

APL_FUNCTIONAL_SYMBOL_QUAD_DEL = ''

APL_FUNCTIONAL_SYMBOL_QUAD_DELTA = ''

APL_FUNCTIONAL_SYMBOL_QUAD_DIAMOND = ''

APL_FUNCTIONAL_SYMBOL_QUAD_DIVIDE = ''

APL_FUNCTIONAL_SYMBOL_QUAD_DOWNWARDS_ARROW = ''

APL_FUNCTIONAL_SYMBOL_QUAD_DOWN_CARET = ''

APL_FUNCTIONAL_SYMBOL_QUAD_EQUAL = ''

APL_FUNCTIONAL_SYMBOL_QUAD_GREATER_THAN = ''

APL_FUNCTIONAL_SYMBOL_QUAD_JOT = ''

APL_FUNCTIONAL_SYMBOL_QUAD_LEFTWARDS_ARROW = ''

APL_FUNCTIONAL_SYMBOL_QUAD_LESS_THAN = ''

APL_FUNCTIONAL_SYMBOL_QUAD_NOT_EQUAL = ''

APL_FUNCTIONAL_SYMBOL_QUAD_QUESTION = ''

APL_FUNCTIONAL_SYMBOL_QUAD_RIGHTWARDS_ARROW = ''

APL_FUNCTIONAL_SYMBOL_QUAD_SLASH = ''

APL_FUNCTIONAL_SYMBOL_QUAD_UPWARDS_ARROW = ''

APL_FUNCTIONAL_SYMBOL_QUAD_UP_CARET = ''

APL_FUNCTIONAL_SYMBOL_QUOTE_QUAD = ''

APL_FUNCTIONAL_SYMBOL_QUOTE_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_RHO = ''

APL_FUNCTIONAL_SYMBOL_RIGHTWARDS_VANE = ''

APL_FUNCTIONAL_SYMBOL_SEMICOLON_UNDERBAR = ''

APL_FUNCTIONAL_SYMBOL_SLASH_BAR = ''

APL_FUNCTIONAL_SYMBOL_SQUISH_QUAD = ''

APL_FUNCTIONAL_SYMBOL_STAR_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_STILE_TILDE = ''

APL_FUNCTIONAL_SYMBOL_TILDE_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_UPWARDS_VANE = ''

APL_FUNCTIONAL_SYMBOL_UP_CARET_TILDE = ''

APL_FUNCTIONAL_SYMBOL_UP_SHOE_JOT = ''

APL_FUNCTIONAL_SYMBOL_UP_TACK_DIAERESIS = ''

APL_FUNCTIONAL_SYMBOL_UP_TACK_JOT = ''

APL_FUNCTIONAL_SYMBOL_UP_TACK_OVERBAR = ''

APL_FUNCTIONAL_SYMBOL_ZILDE = ''

ARC = ''

BELL_SYMBOL = ''

BENZENE_RING = ''

BENZENE_RING_WITH_CIRCLE = ''

BLACK_CIRCLE_FOR_RECORD = ''

BLACK_DOWN_POINTING_DOUBLE_TRIANGLE = ''

BLACK_LEFT_POINTING_DOUBLE_TRIANGLE = ''

BLACK_LEFT_POINTING_DOUBLE_TRIANGLE_WITH_VERTICAL_BAR = ''

BLACK_MEDIUM_DOWN_POINTING_TRIANGLE = ''

BLACK_MEDIUM_LEFT_POINTING_TRIANGLE = ''

BLACK_MEDIUM_RIGHT_POINTING_TRIANGLE = ''

BLACK_MEDIUM_UP_POINTING_TRIANGLE = ''

BLACK_RIGHT_POINTING_DOUBLE_TRIANGLE = ''

BLACK_RIGHT_POINTING_DOUBLE_TRIANGLE_WITH_VERTICAL_BAR = ''

BLACK_RIGHT_POINTING_TRIANGLE_WITH_DOUBLE_VERTICAL_BAR = ''

BLACK_SQUARE_FOR_STOP = ''

BLACK_UP_POINTING_DOUBLE_TRIANGLE = ''

BOTTOM_CURLY_BRACKET = ''

BOTTOM_HALF_INTEGRAL = ''

BOTTOM_LEFT_CORNER = ''

BOTTOM_LEFT_CROP = ''

BOTTOM_PARENTHESIS = ''

BOTTOM_RIGHT_CORNER = ''

BOTTOM_RIGHT_CROP = ''

BOTTOM_SQUARE_BRACKET = ''

BOTTOM_SQUARE_BRACKET_OVER_TOP_SQUARE_BRACKET = ''

BOTTOM_TORTOISE_SHELL_BRACKET = ''

BROKEN_CIRCLE_WITH_NORTHWEST_ARROW = ''

CIRCLED_HORIZONTAL_BAR_WITH_NOTCH = ''

CIRCLED_TRIANGLE_DOWN = ''

CLEAR_SCREEN_SYMBOL = ''

COMPOSITION_SYMBOL = ''

CONICAL_TAPER = ''

CONTINUOUS_UNDERLINE_SYMBOL = ''

COUNTERBORE = ''

COUNTERSINK = ''

CURLY_BRACKET_EXTENSION = ''

CYLINDRICITY = ''

DECIMAL_EXPONENT_SYMBOL = ''

DECIMAL_SEPARATOR_KEY_SYMBOL = ''

DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL = ''

DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL_WITH_CIRCLE = ''

DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL_WITH_TRIANGLE = ''

DENTISTRY_SYMBOL_LIGHT_DOWN_AND_HORIZONTAL_WITH_WAVE = ''

DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL = ''

DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_WITH_CIRCLE = ''

DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_WITH_TRIANGLE = ''

DENTISTRY_SYMBOL_LIGHT_UP_AND_HORIZONTAL_WITH_WAVE = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_BOTTOM_LEFT = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_BOTTOM_RIGHT = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_TOP_LEFT = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_TOP_RIGHT = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_AND_WAVE = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_WITH_CIRCLE = ''

DENTISTRY_SYMBOL_LIGHT_VERTICAL_WITH_TRIANGLE = ''

DIAMETER_SIGN = ''

DIMENSION_ORIGIN = ''

DIRECT_CURRENT_SYMBOL_FORM_TWO = ''

DISCONTINUOUS_UNDERLINE_SYMBOL = ''

DOUBLE_VERTICAL_BAR = ''

DOWN_ARROWHEAD = ''

EARTH_GROUND = ''

EJECT_SYMBOL = ''

ELECTRICAL_INTERSECTION = ''

ELECTRIC_ARROW = ''

EMPHASIS_SYMBOL = ''

ENTER_SYMBOL = ''

ERASE_TO_THE_LEFT = ''

ERASE_TO_THE_RIGHT = ''

FLATNESS = ''

FROWN = ''

FUSE = ''

HELM_SYMBOL = ''

HORIZONTAL_LINE_EXTENSION = ''

HORIZONTAL_SCAN_LINE_1 = ''

HORIZONTAL_SCAN_LINE_3 = ''

HORIZONTAL_SCAN_LINE_7 = ''

HORIZONTAL_SCAN_LINE_9 = '_'

HOURGLASS = ''

HOURGLASS_WITH_FLOWING_SAND = ''

HOUSE = ''

HYSTERESIS_SYMBOL = ''

INSERTION_SYMBOL = ''

INTEGRAL_EXTENSION = ''

KEYBOARD = ''

LEFT_CEILING = ''

LEFT_CURLY_BRACKET_LOWER_HOOK = ''

LEFT_CURLY_BRACKET_MIDDLE_PIECE = ''

LEFT_CURLY_BRACKET_UPPER_HOOK = ''

LEFT_FLOOR = ''

LEFT_PARENTHESIS_EXTENSION = ''

LEFT_PARENTHESIS_LOWER_HOOK = ''

LEFT_PARENTHESIS_UPPER_HOOK = ''

LEFT_POINTING_ANGLE_BRACKET = '〈'

LEFT_SQUARE_BRACKET_EXTENSION = ''

LEFT_SQUARE_BRACKET_LOWER_CORNER = ''

LEFT_SQUARE_BRACKET_UPPER_CORNER = ''

LEFT_VERTICAL_BOX_LINE = ''

METRICAL_BREVE = ''

METRICAL_LONG_OVER_SHORT = ''

METRICAL_LONG_OVER_TWO_SHORTS = ''

METRICAL_PENTASEME = ''

METRICAL_SHORT_OVER_LONG = ''

METRICAL_TETRASEME = ''

METRICAL_TRISEME = ''

METRICAL_TWO_SHORTS_JOINED = ''

METRICAL_TWO_SHORTS_OVER_LONG = ''

MONOSTABLE_SYMBOL = ''

NEXT_PAGE = ''

NOT_CHECK_MARK = ''

OBSERVER_EYE_SYMBOL = ''

OPEN_CIRCUIT_OUTPUT_H_TYPE_SYMBOL = ''

OPEN_CIRCUIT_OUTPUT_L_TYPE_SYMBOL = ''

OPTION_KEY = ''

PASSIVE_PULL_DOWN_OUTPUT_SYMBOL = ''

PASSIVE_PULL_UP_OUTPUT_SYMBOL = ''

PERSPECTIVE = ''

PLACE_OF_INTEREST_SIGN = ''

POSITION_INDICATOR = ''

POWER_ON_OFF_SYMBOL = ''

POWER_ON_SYMBOL = ''

POWER_SLEEP_SYMBOL = ''

POWER_SYMBOL = ''

PREVIOUS_PAGE = ''

PRINT_SCREEN_SYMBOL = ''

PROJECTIVE = ''

RADICAL_SYMBOL_BOTTOM = ''

RETURN_SYMBOL = ''

REVERSED_NOT_SIGN = ''

RIGHT_ANGLE_WITH_DOWNWARDS_ZIGZAG_ARROW = ''

RIGHT_CEILING = ''

RIGHT_CURLY_BRACKET_LOWER_HOOK = ''

RIGHT_CURLY_BRACKET_MIDDLE_PIECE = ''

RIGHT_CURLY_BRACKET_UPPER_HOOK = ''

RIGHT_FLOOR = ''

RIGHT_PARENTHESIS_EXTENSION = ''

RIGHT_PARENTHESIS_LOWER_HOOK = ''

RIGHT_PARENTHESIS_UPPER_HOOK = ''

RIGHT_POINTING_ANGLE_BRACKET = '〉'

RIGHT_SQUARE_BRACKET_EXTENSION = ''

RIGHT_SQUARE_BRACKET_LOWER_CORNER = ''

RIGHT_SQUARE_BRACKET_UPPER_CORNER = ''

RIGHT_VERTICAL_BOX_LINE = ''

SECTOR = ''

SEGMENT = ''

SHOULDERED_OPEN_BOX = ''

SLOPE = ''

SMILE = ''

SOFTWARE_FUNCTION_SYMBOL = ''

SQUARE_FOOT = ''

SQUARE_LOZENGE = ''

STOPWATCH = ''

STRAIGHTNESS = ''

SUMMATION_BOTTOM = ''

SUMMATION_TOP = ''

SYMMETRY = ''

TELEPHONE_RECORDER = ''

TIMER_CLOCK = ''

TOP_CURLY_BRACKET = ''

TOP_HALF_INTEGRAL = ''

TOP_LEFT_CORNER = ''

TOP_LEFT_CROP = ''

TOP_PARENTHESIS = ''

TOP_RIGHT_CORNER = ''

TOP_RIGHT_CROP = ''

TOP_SQUARE_BRACKET = ''

TOP_TORTOISE_SHELL_BRACKET = ''

TOTAL_RUNOUT = ''

TURNED_NOT_SIGN = ''

UNDO_SYMBOL = ''

UPPER_LEFT_OR_LOWER_RIGHT_CURLY_BRACKET_SECTION = ''

UPPER_RIGHT_OR_LOWER_LEFT_CURLY_BRACKET_SECTION = ''

UP_ARROWHEAD = ''

UP_ARROWHEAD_BETWEEN_TWO_HORIZONTAL_BARS = ''

VERTICAL_LINE_EXTENSION = ''

VERTICAL_LINE_WITH_MIDDLE_DOT = ''

VIEWDATA_SQUARE = ''

WATCH = ''

WAVY_LINE = ''

WHITE_SQUARE_WITH_CENTRE_VERTICAL_LINE = ''

WHITE_TRAPEZIUM = ''

X_IN_A_RECTANGLE_BOX = ''

Models

class pygamelib.assets.graphics.**Models**

Bases: object

List of models (emojis by unicode denomination)

Models are filtered emojis. This class does not map the entire specification.

Models replaces the previous Sprites class. Renaming that class is necessary with the introduction of a real Sprite class in the GFX module.

This class contains 1328 emojis (this is not the full list). All emoji codes come from: https://unicode.org/emoji/charts/full_emoji_list.html Additional emojis can be added by codes.

The complete list of aliased emojis is:

- GRINNING_FACE =
- GRINNING_FACE_WITH_BIG_EYES =
- GRINNING_FACE_WITH_SMILING_EYES =
- BEAMING_FACE_WITH_SMILING_EYES =
- GRINNING_SQUINTING_FACE =
- GRINNING_FACE_WITH_SWEAT =
- ROLLING_ON_THE_FLOOR_LAUGHING =

- FACE_WITH_TEARS_OF_JOY =
- SLIGHTLY_SMILING_FACE =
- UPSIDE_DOWN_FACE =
- WINKING_FACE =
- SMILING_FACE_WITH_SMILING_EYES =
- SMILING_FACE_WITH_HALO =
- SMILING_FACE_WITH_HEARTS =
- SMILING_FACE_WITH_HEART_EYES =
- STAR_STRUCK =
- FACE_BLOWING_A_KISS =
- KISSING_FACE =
- SMILING_FACE =
- KISSING_FACE_WITH_CLOSED_EYES =
- KISSING_FACE_WITH_SMILING_EYES =
- SMILING_FACE_WITH_TEAR =
- FACE_SAVORING_FOOD =
- FACE_WITH_TONGUE =
- WINKING_FACE_WITH_TONGUE =
- ZANY_FACE =
- SQUINTING_FACE_WITH_TONGUE =
- MONEY_MOUTH_FACE =
- HUGGING_FACE =
- FACE_WITH_HAND_OVER_MOUTH =
- SHUSHING_FACE =
- THINKING_FACE =
- ZIPPER_MOUTH_FACE =
- FACE_WITH_RAISED_EYEBROW =
- NEUTRAL_FACE =
- EXPRESSIONLESS_FACE =
- FACE_WITHOUT_MOUTH =
- SMIRKING_FACE =
- UNAMUSED_FACE =
- FACE_WITH_ROLLING_EYES =
- GRIMACING_FACE =
- LYING_FACE =
- RELIEVED_FACE =

- PENSIVE_FACE =
- SLEEPY_FACE =
- DROOLING_FACE =
- SLEEPING_FACE =
- FACE_WITH_MEDICAL_MASK =
- FACE_WITH_THERMOMETER =
- FACE_WITH_HEAD_BANDAGE =
- NAUSEATED_FACE =
- FACE_VOMITING =
- SNEEZING_FACE =
- HOT_FACE =
- COLD_FACE =
- WOOZY_FACE =
- DIZZY_FACE =
- EXPLODING_HEAD =
- COWBOY_HAT_FACE =
- PARTYING_FACE =
- DISGUISED_FACE =
- SMILING_FACE_WITH_SUNGLASSES =
- NERD_FACE =
- FACE_WITH_MONOCLE =
- CONFUSED_FACE =
- WORRIED_FACE =
- SLIGHTLY_FROWNING_FACE =
- FROWNING_FACE =
- FACE_WITH_OPEN_MOUTH =
- HUSHED_FACE =
- ASTONISHED_FACE =
- FLUSHED_FACE =
- PLEADING_FACE =
- FROWNING_FACE_WITH_OPEN_MOUTH =
- ANGUISHED_FACE =
- FEARFUL_FACE =
- ANXIOUS_FACE_WITH_SWEAT =
- SAD_BUT_RELIEVED_FACE =
- CRYING_FACE =

- LOUDLY_CRYING_FACE =
- FACE_SCREAMING_IN_FEAR =
- CONFOUNDED_FACE =
- PERSEVERING_FACE =
- DISAPPOINTED_FACE =
- DOWNCAST_FACE_WITH_SWEAT =
- WEARY_FACE =
- TIRED_FACE =
- YAWNING_FACE =
- FACE_WITH_STEAM_FROM_NOSE =
- POUTING_FACE =
- ANGRY_FACE =
- FACE_WITH_SYMBOLS_ON_MOUTH =
- SMILING_FACE_WITH_HORNS =
- ANGRY_FACE_WITH_HORNS =
- SKULL =
- SKULL_AND_CROSSBONES =
- PILE_OF_POO =
- CLOWN_FACE =
- OGRE =
- GOBLIN =
- GHOST =
- ALIEN =
- ALIEN_MONSTER =
- ROBOT =
- GRINNING_CAT =
- GRINNING_CAT_WITH_SMILING_EYES =
- CAT_WITH_TEARS_OF_JOY =
- SMILING_CAT_WITH_HEART_EYES =
- CAT_WITH_WRY_SMILE =
- KISSING_CAT =
- WEARY_CAT =
- CRYING_CAT =
- POUTING_CAT =
- SEE_NO_EVIL_MONKEY =
- HEAR_NO_EVIL_MONKEY =

- SPEAK_NO_EVIL_MONKEY =
- KISS_MARK =
- LOVE_LETTER =
- HEART_WITH_ARROW =
- HEART_WITH_RIBBON =
- SPARKLING_HEART =
- GROWING_HEART =
- BEATING_HEART =
- REVOLVING_HEARTS =
- TWO_HEARTS =
- HEART_DECORATION =
- HEART_EXCLAMATION =
- BROKEN_HEART =
- RED_HEART =
- ORANGE_HEART =
- YELLOW_HEART =
- GREEN_HEART =
- BLUE_HEART =
- PURPLE_HEART =
- BROWN_HEART =
- BLACK_HEART =
- WHITE_HEART =
- HUNDRED_POINTS =
- ANGER_SYMBOL =
- COLLISION =
- DIZZY =
- SWEAT_DROPLETS =
- DASHING_AWAY =
- HOLE =
- BOMB =
- SPEECH_BALLOON =
- LEFT_SPEECH_BUBBLE =
- RIGHT_ANGER_BUBBLE =
- THOUGHT_BALLOON =
- ZZZ =
- WAVING_HAND =

- RAISED_BACK_OF_HAND =
- HAND_WITH_FINGERS_SPLAYED =
- RAISED_HAND =
- VULCAN_SALUTE =
- OK_HAND =
- PINCHED_FINGERS =
- PINCHING_HAND =
- VICTORY_HAND =
- CROSSED_FINGERS =
- LOVE_YOU_GESTURE =
- SIGN_OF_THE_HORNS =
- CALL_ME_HAND =
- BACKHAND_INDEX_POINTING_LEFT =
- BACKHAND_INDEX_POINTING_RIGHT =
- BACKHAND_INDEX_POINTING_UP =
- MIDDLE_FINGER =
- BACKHAND_INDEX_POINTING_DOWN =
- INDEX_POINTING_UP =
- THUMBS_UP =
- THUMBS_DOWN =
- RAISED_FIST =
- ONCOMING_FIST =
- LEFT_FACING_FIST =
- RIGHT_FACING_FIST =
- CLAPPING_HANDS =
- RAISING_HANDS =
- OPEN_HANDS =
- PALMS_UP_TOGETHER =
- HANDSHAKE =
- FOLDED_HANDS =
- WRITING_HAND =
- NAIL_POLISH =
- SELFIE =
- FLEXED_BICEPS =
- MECHANICAL_ARM =
- MECHANICAL_LEG =

- LEG =
- FOOT =
- EAR =
- EAR_WITH_HEARING_AID =
- NOSE =
- BRAIN =
- ANATOMICAL_HEART =
- LUNGS =
- TOOTH =
- BONE =
- EYES =
- EYE =
- TONGUE =
- MOUTH =
- BABY =
- CHILD =
- BOY =
- GIRL =
- PERSON =
- PERSON_BLOND_HAIR =
- MAN =
- MAN_BEARD =
- WOMAN =
- OLDER_PERSON =
- OLD_MAN =
- OLD_WOMAN =
- PERSON_FROWNING =
- PERSON_POUTING =
- PERSON_GESTURING_NO =
- PERSON_GESTURING_OK =
- PERSON_TIPPING_HAND =
- PERSON_RAISING_HAND =
- DEAF_PERSON =
- PERSON_BOWING =
- PERSON_FACEPALMING =
- PERSON_SHRUGGING =

- POLICE_OFFICER =
- DETECTIVE =
- GUARD =
- NINJA =
- CONSTRUCTION_WORKER =
- PRINCE =
- PRINCESS =
- PERSON_WEARING_TURBAN =
- PERSON_WITH_SKULLCAP =
- WOMAN_WITH_HEADSCARF =
- PERSON_IN_TUXEDO =
- PERSON_WITH_VEIL =
- PREGNANT_WOMAN =
- BREAST_FEEDING =
- BABY_ANGEL =
- SANTA_CLAUS =
- MRS_CLAUS =
- SUPERHERO =
- SUPERVILLAIN =
- MAGE =
- FAIRY =
- VAMPIRE =
- MERPERSON =
- ELF =
- GENIE =
- ZOMBIE =
- PERSON_GETTING_MASSAGE =
- PERSON_GETTING_HAIRCUT =
- PERSON_WALKING =
- PERSON_STANDING =
- PERSON_KNEELING =
- PERSON_RUNNING =
- WOMAN_DANCING =
- MAN_DANCING =
- PERSON_IN_SUIT_LEVITATING =
- PEOPLE_WITH_BUNNY_EARS =

- PERSON_IN_STEAMY_ROOM =
- PERSON_CLIMBING =
- PERSON_FENCING =
- HORSE_RACING =
- SKIER =
- SNOWBOARDER =
- PERSON_GOLFING =
- PERSON_SURFING =
- PERSON_ROWING_BOAT =
- PERSON_SWIMMING =
- PERSON_BOUNCING_BALL =
- PERSON_LIFTING_WEIGHTS =
- PERSON_BIKING =
- PERSON_MOUNTAIN_BIKING =
- PERSON_CARTWHEELING =
- PEOPLE_WRESTLING =
- PERSON_PLAYING_WATER_POLO =
- PERSON_PLAYING_HANDBALL =
- PERSON_JUGGLING =
- PERSON_IN_LOTUS_POSITION =
- PERSON_TAKING_BATH =
- PERSON_IN_BED =
- WOMEN_HOLDING_HANDS =
- WOMAN_AND_MAN_HOLDING_HANDS =
- MEN_HOLDING_HANDS =
- KISS =
- COUPLE_WITH_HEART =
- FAMILY =
- SPEAKING_HEAD =
- BUST_IN_SILHOUETTE =
- BUSTS_IN_SILHOUETTE =
- PEOPLE_HUGGING =
- FOOTPRINTS =
- LIGHT_SKIN_TONE =
- MEDIUM_LIGHT_SKIN_TONE =
- MEDIUM_SKIN_TONE =

- MEDIUM_DARK_SKIN_TONE =
- DARK_SKIN_TONE =
- RED_HAIR =
- CURLY_HAIR =
- WHITE_HAIR =
- BALD =
- MONKEY_FACE =
- MONKEY =
- GORILLA =
- ORANGUTAN =
- DOG_FACE =
- DOG =
- GUIDE_DOG =
- POODLE =
- WOLF =
- FOX =
- RACCOON =
- CAT_FACE =
- CAT =
- LION =
- TIGER_FACE =
- TIGER =
- LEOPARD =
- HORSE_FACE =
- HORSE =
- UNICORN =
- ZEBRA =
- DEER =
- BISON =
- COW_FACE =
- OX =
- WATER_BUFFALO =
- COW =
- PIG_FACE =
- PIG =
- BOAR =

- PIG_NOSE =
- RAM =
- EWE =
- GOAT =
- CAMEL =
- TWO_HUMP_CAMEL =
- LLAMA =
- GIRAFFE =
- ELEPHANT =
- MAMMOTH =
- RHINOCEROS =
- HIPPOPOTAMUS =
- MOUSE_FACE =
- MOUSE =
- RAT =
- HAMSTER =
- RABBIT_FACE =
- RABBIT =
- CHIPMUNK =
- BEAVER =
- HEDGEHOG =
- BAT =
- BEAR =
- KOALA =
- PANDA =
- SLOTH =
- OTTER =
- SKUNK =
- KANGAROO =
- BADGER =
- PAW_PRINTS =
- TURKEY =
- CHICKEN =
- ROOSTER =
- HATCHING_CHICK =
- BABY_CHICK =

- FRONT_FACING_BABY_CHICK =
- BIRD =
- PENGUIN =
- DOVE =
- EAGLE =
- DUCK =
- SWAN =
- OWL =
- DODO =
- FEATHER =
- FLAMINGO =
- PEACOCK =
- PARROT =
- FROG =
- CROCODILE =
- TURTLE =
- LIZARD =
- SNAKE =
- DRAGON_FACE =
- DRAGON =
- SAUROPOD =
- T_REX =
- SPOUTING_WHALE =
- WHALE =
- DOLPHIN =
- SEAL =
- FISH =
- TROPICAL_FISH =
- BLOWFISH =
- SHARK =
- OCTOPUS =
- SPIRAL_SHELL =
- SNAIL =
- BUTTERFLY =
- BUG =
- ANT =

- HONEYBEE =
- BEETLE =
- LADY_BEETLE =
- CRICKET =
- COCKROACH =
- SPIDER =
- SPIDER_WEB =
- SCORPION =
- MOSQUITO =
- FLY =
- WORM =
- MICROBE =
- BOUQUET =
- CHERRY_BLOSSOM =
- WHITE_FLOWER =
- ROSETTE =
- ROSE =
- WILTED_FLOWER =
- HIBISCUS =
- SUNFLOWER =
- BLOSSOM =
- TULIP =
- SEEDLING =
- POTTED_PLANT =
- EVERGREEN_TREE =
- DECIDUOUS_TREE =
- PALM_TREE =
- CACTUS =
- SHEAF_OF_RICE =
- HERB =
- SHAMROCK =
- FOUR_LEAF_CLOVER =
- MAPLE_LEAF =
- FALLEN_LEAF =
- LEAF_FLUTTERING_IN_WIND =
- GRAPES =

- MELON =
- WATERMELON =
- TANGERINE =
- LEMON =
- BANANA =
- PINEAPPLE =
- MANGO =
- RED_APPLE =
- GREEN_APPLE =
- PEAR =
- PEACH =
- CHERRIES =
- STRAWBERRY =
- BLUEBERRIES =
- KIWI_FRUIT =
- TOMATO =
- OLIVE =
- COCONUT =
- AVOCADO =
- EGGPLANT =
- POTATO =
- CARROT =
- EAR_OF_CORN =
- HOT_PEPPER =
- BELL_PEPPER =
- CUCUMBER =
- LEAFY_GREEN =
- BROCCOLI =
- GARLIC =
- ONION =
- MUSHROOM =
- PEANUTS =
- CHESTNUT =
- BREAD =
- CROISSANT =
- BAGUETTE_BREAD =

- FLATBREAD =
- PRETZEL =
- BAGEL =
- PANCAKES =
- WAFFLE =
- CHEESE_WEDGE =
- MEAT_ON_BONE =
- POULTRY_LEG =
- CUT_OF_MEAT =
- BACON =
- HAMBURGER =
- FRENCH_FRIES =
- PIZZA =
- HOT_DOG =
- SANDWICH =
- TACO =
- BURRITO =
- TAMALE =
- STUFFED_FLATBREAD =
- FALAFEL =
- EGG =
- COOKING =
- SHALLOW_PAN_OF_FOOD =
- POT_OF_FOOD =
- FONDUE =
- BOWL_WITH_SPOON =
- GREEN_SALAD =
- POPCORN =
- BUTTER =
- SALT =
- CANNED_FOOD =
- BENTO_BOX =
- RICE_CRACKER =
- RICE_BALL =
- COOKED_RICE =
- CURRY_RICE =

- STEAMING_BOWL =
- SPAGHETTI =
- ROASTED_SWEET_POTATO =
- ODEN =
- SUSHI =
- FRIED_SHRIMP =
- FISH_CAKE_WITH_SWIRL =
- MOON_CAKE =
- DANGO =
- DUMPLING =
- FORTUNE_COOKIE =
- TAKEOUT_BOX =
- CRAB =
- LOBSTER =
- SHRIMP =
- SQUID =
- OYSTER =
- SOFT_ICE_CREAM =
- SHAVED_ICE =
- ICE_CREAM =
- DOUGHNUT =
- COOKIE =
- BIRTHDAY_CAKE =
- SHORTCAKE =
- CUPCAKE =
- PIE =
- CHOCOLATE_BAR =
- CANDY =
- LOLLIPOP =
- CUSTARD =
- HONEY_POT =
- BABY_BOTTLE =
- GLASS_OF_MILK =
- HOT_BEVERAGE =
- TEAPOT =
- TEACUP_WITHOUT_HANDLE =

- SAKE =
- BOTTLE_WITH_POPPING_CORK =
- WINE_GLASS =
- COCKTAIL_GLASS =
- TROPICAL_DRINK =
- BEER_MUG =
- CLINKING_BEER_MUGS =
- CLINKING_GLASSES =
- TUMBLER_GLASS =
- CUP_WITH_STRAW =
- BUBBLE_TEA =
- BEVERAGE_BOX =
- MATE =
- ICE =
- CHOPSTICKS =
- FORK_AND_KNIFE_WITH_PLATE =
- FORK_AND_KNIFE =
- SPOON =
- KITCHEN_KNIFE =
- AMPHORA =
- GLOBE_SHOWING_EUROPE_AFRICA =
- GLOBE_SHOWING_AMERICAS =
- GLOBE_SHOWING_ASIA_AUSTRALIA =
- GLOBE_WITH_MERIDIANS =
- WORLD_MAP =
- MAP_OF_JAPAN =
- COMPASS =
- SNOW_CAPPED_MOUNTAIN =
- MOUNTAIN =
- VOLCANO =
- MOUNT_FUJI =
- CAMPING =
- BEACH_WITH_UMBRELLA =
- DESERT =
- DESERT_ISLAND =
- NATIONAL_PARK =

- STADIUM =
- CLASSICAL_BUILDING =
- BUILDING_CONSTRUCTION =
- BRICK =
- ROCK =
- WOOD =
- HUT =
- HOUSES =
- DERELICT_HOUSE =
- HOUSE =
- HOUSE_WITH_GARDEN =
- OFFICE_BUILDING =
- JAPANESE_POST_OFFICE =
- POST_OFFICE =
- HOSPITAL =
- BANK =
- HOTEL =
- LOVE_HOTEL =
- CONVENIENCE_STORE =
- SCHOOL =
- DEPARTMENT_STORE =
- FACTORY =
- JAPANESE_CASTLE =
- CASTLE =
- WEDDING =
- TOKYO_TOWER =
- STATUE_OF_LIBERTY =
- CHURCH =
- MOSQUE =
- HINDU_TEMPLE =
- SYNAGOGUE =
- SHINTO_SHRINE =
- KAABA =
- FOUNTAIN =
- TENT =
- FOGGY =

- NIGHT_WITH_STARS =
- CITYSCAPE =
- SUNRISE_OVER_MOUNTAINS =
- SUNRISE =
- CITYSCAPE_AT_DUSK =
- SUNSET =
- BRIDGE_AT_NIGHT =
- HOT_SPRINGS =
- CAROUSEL_HORSE =
- FERRIS_WHEEL =
- ROLLER_COASTER =
- BARBER_POLE =
- CIRCUS_TENT =
- LOCOMOTIVE =
- RAILWAY_CAR =
- HIGH_SPEED_TRAIN =
- BULLET_TRAIN =
- TRAIN =
- METRO =
- LIGHT_RAIL =
- STATION =
- TRAM =
- MONORAIL =
- MOUNTAIN_RAILWAY =
- TRAM_CAR =
- BUS =
- ONCOMING_BUS =
- TROLLEYBUS =
- MINIBUS =
- AMBULANCE =
- FIRE_ENGINE =
- POLICE_CAR =
- ONCOMING_POLICE_CAR =
- TAXI =
- ONCOMING_TAXI =
- AUTOMOBILE =

- ONCOMING_AUTOMOBILE =
- SPORT_UTILITY_VEHICLE =
- PICKUP_TRUCK =
- DELIVERY_TRUCK =
- ARTICULATED_LORRY =
- TRACTOR =
- RACING_CAR =
- MOTORCYCLE =
- MOTOR_SCOOTER =
- MANUAL_WHEELCHAIR =
- MOTORIZED_WHEELCHAIR =
- AUTO_RICKSHAW =
- BICYCLE =
- KICK_SCOOTER =
- SKATEBOARD =
- ROLLER_SKATE =
- BUS_STOP =
- MOTORWAY =
- RAILWAY_TRACK =
- OIL_DRUM =
- FUEL_PUMP =
- POLICE_CAR_LIGHT =
- HORIZONTAL_TRAFFIC_LIGHT =
- VERTICAL_TRAFFIC_LIGHT =
- STOP_SIGN =
- CONSTRUCTION =
- ANCHOR =
- SAILBOAT =
- CANOE =
- SPEEDBOAT =
- PASSENGER_SHIP =
- FERRY =
- MOTOR_BOAT =
- SHIP =
- AIRPLANE =
- SMALL_AIRPLANE =

- AIRPLANE_DEPARTURE =
- AIRPLANE_ARRIVAL =
- PARACHUTE =
- SEAT =
- HELICOPTER =
- SUSPENSION_RAILWAY =
- MOUNTAIN_CABLEWAY =
- AERIAL_TRAMWAY =
- SATELLITE =
- ROCKET =
- FLYING_SAUCER =
- BELLHOP_BELL =
- LUGGAGE =
- HOURGLASS_DONE =
- HOURGLASS_NOT_DONE =
- WATCH =
- ALARM_CLOCK =
- STOPWATCH =
- TIMER_CLOCK =
- MANTELPIECE_CLOCK =
- TWELVE_OCLOCK =
- TWELVE_THIRTY =
- ONE_OCLOCK =
- ONE_THIRTY =
- TWO_OCLOCK =
- TWO_THIRTY =
- THREE_OCLOCK =
- THREE_THIRTY =
- FOUR_OCLOCK =
- FOUR_THIRTY =
- FIVE_OCLOCK =
- FIVE_THIRTY =
- SIX_OCLOCK =
- SIX_THIRTY =
- SEVEN_OCLOCK =
- SEVEN_THIRTY =

- EIGHT_OCLOCK =
- EIGHT_THIRTY =
- NINE_OCLOCK =
- NINE_THIRTY =
- TEN_OCLOCK =
- TEN_THIRTY =
- ELEVEN_OCLOCK =
- ELEVEN_THIRTY =
- NEW_MOON =
- WAXING_CRESCENT_MOON =
- FIRST_QUARTER_MOON =
- WAXING_GIBBOUS_MOON =
- FULL_MOON =
- WANING_GIBBOUS_MOON =
- LAST_QUARTER_MOON =
- WANING_CRESCENT_MOON =
- CRESCENT_MOON =
- NEW_MOON_FACE =
- FIRST_QUARTER_MOON_FACE =
- LAST_QUARTER_MOON_FACE =
- THERMOMETER =
- SUN =
- FULL_MOON_FACE =
- SUN_WITH_FACE =
- RINGED_PLANET =
- STAR =
- GLOWING_STAR =
- SHOOTING_STAR =
- MILKY_WAY =
- CLOUD =
- SUN_BEHIND_CLOUD =
- CLOUD_WITH_LIGHTNING_AND_RAIN =
- SUN_BEHIND_SMALL_CLOUD =
- SUN_BEHIND_LARGE_CLOUD =
- SUN_BEHIND_RAIN_CLOUD =
- CLOUD_WITH_RAIN =

- CLOUD_WITH_SNOW =
- CLOUD_WITH_LIGHTNING =
- TORNADO =
- FOG =
- WIND_FACE =
- CYCLONE =
- RAINBOW =
- CLOSED_UMBRELLA =
- UMBRELLA =
- UMBRELLA_WITH_RAIN_DROPS =
- UMBRELLA_ON_GROUND =
- HIGH_VOLTAGE =
- SNOWFLAKE =
- SNOWMAN =
- SNOWMAN_WITHOUT_SNOW =
- COMET =
- FIRE =
- DROPLET =
- WATER_WAVE =
- JACK_O_LANTERN =
- CHRISTMAS_TREE =
- FIREWORKS =
- SPARKLER =
- FIRECRACKER =
- SPARKLES =
- BALLOON =
- PARTY_POPPER =
- CONFETTI_BALL =
- TANABATA_TREE =
- PINE_DECORATION =
- JAPANESE_DOLLS =
- CARP_STREAMER =
- WIND_CHIME =
- MOON_VIEWING_CEREMONY =
- RED_ENVELOPE =
- RIBBON =

- WRAPPED_GIFT =
- REMINDER_RIBBON =
- ADMISSION_TICKETS =
- TICKET =
- MILITARY_MEDAL =
- TROPHY =
- SPORTS_MEDAL =
- FIRST_PLACE_MEDAL =
- SECOND_PLACE_MEDAL =
- THIRD_PLACE_MEDAL =
- SOCCER_BALL =
- BASEBALL =
- SOFTBALL =
- BASKETBALL =
- VOLLEYBALL =
- AMERICAN_FOOTBALL =
- RUGBY_FOOTBALL =
- TENNIS =
- FLYING_DISC =
- BOWLING =
- CRICKET_GAME =
- FIELD_HOCKEY =
- ICE_HOCKEY =
- LACROSSE =
- PING_PONG =
- BADMINTON =
- BOXING_GLOVE =
- MARTIAL_ARTS_UNIFORM =
- GOAL_NET =
- FLAG_IN_HOLE =
- ICE_SKATE =
- FISHING_POLE =
- DIVING_MASK =
- RUNNING_SHIRT =
- SKIS =
- SLED =

- CURLING_STONE =
- DIRECT_HIT =
- YO_YO =
- KITE =
- BALL =
- CRYSTAL_BALL =
- MAGIC_WAND =
- NAZAR_AMULET =
- VIDEO_GAME =
- JOYSTICK =
- SLOT_MACHINE =
- GAME_DIE =
- PUZZLE_PIECE =
- TEDDY_BEAR =
- PIñATA =
- NESTING_DOLLS =
- SPADE_SUIT =
- HEART_SUIT =
- DIAMOND_SUIT =
- CLUB_SUIT =
- CHESS_PAWN =
- JOKER =
- MAHJONG_RED_DRAGON =
- FLOWER_PLAYING_CARDS =
- PERFORMING_ARTS =
- FRAMED_PICTURE =
- ARTIST_PALETTE =
- THREAD =
- SEWING_NEEDLE =
- YARN =
- KNOT =
- GLASSES =
- SUNGLASSES =
- GOGGLES =
- LAB_COAT =
- SAFETY_VEST =

- NECKTIE =
- T_SHIRT =
- JEANS =
- SCARF =
- GLOVES =
- COAT =
- SOCKS =
- DRESS =
- KIMONO =
- SARI =
- ONE_PIECE_SWIMSUIT =
- BRIEFS =
- SHORTS =
- BIKINI =
- WOMANS_CLOTHES =
- PURSE =
- HANDBAG =
- CLUTCH_BAG =
- SHOPPING_BAGS =
- BACKPACK =
- THONG_SANDAL =
- MANS_SHOE =
- RUNNING_SHOE =
- HIKING_BOOT =
- FLAT_SHOE =
- HIGH_HEELED_SHOE =
- WOMANS_SANDAL =
- BALLET_SHOES =
- WOMANS_BOOT =
- CROWN =
- WOMANS_HAT =
- TOP_HAT =
- GRADUATION_CAP =
- BILLED_CAP =
- MILITARY_HELMET =
- RESCUE_WORKERS_HELMET =

- PRAYER_BEADS =
- LIPSTICK =
- RING =
- GEM_STONE =
- MUTED_SPEAKER =
- SPEAKER_LOW_VOLUME =
- SPEAKER_MEDIUM_VOLUME =
- SPEAKER_HIGH_VOLUME =
- LOUDSPEAKER =
- MEGAPHONE =
- POSTAL_HORN =
- BELL =
- BELL_WITH_SLASH =
- MUSICAL_SCORE =
- MUSICAL_NOTE =
- MUSICAL_NOTES =
- STUDIO_MICROPHONE =
- LEVEL_SLIDER =
- CONTROL_KNOBS =
- MICROPHONE =
- HEADPHONE =
- RADIO =
- SAXOPHONE =
- ACCORDION =
- GUITAR =
- MUSICAL_KEYBOARD =
- TRUMPET =
- VIOLIN =
- BANJO =
- DRUM =
- LONG_DRUM =
- MOBILE_PHONE =
- MOBILE_PHONE_WITH_ARROW =
- TELEPHONE =
- TELEPHONE_RECEIVER =
- PAGER =

- FAX_MACHINE =
- BATTERY =
- ELECTRIC_PLUG =
- LAPTOP =
- DESKTOP_COMPUTER =
- PRINTER =
- KEYBOARD =
- COMPUTER_MOUSE =
- TRACKBALL =
- COMPUTER_DISK =
- FLOPPY_DISK =
- OPTICAL_DISK =
- DVD =
- ABACUS =
- MOVIE_CAMERA =
- FILM_FRAMES =
- FILM_PROJECTOR =
- CLAPPER_BOARD =
- TELEVISION =
- CAMERA =
- CAMERA_WITH_FLASH =
- VIDEO_CAMERA =
- VIDEOCASSETTE =
- MAGNIFYING_GLASS_TILTED_LEFT =
- MAGNIFYING_GLASS_TILTED_RIGHT =
- CANDLE =
- LIGHT_BULB =
- FLASHLIGHT =
- RED_PAPER_LANTERN =
- DIYA_LAMP =
- NOTEBOOK_WITH_DECORATIVE_COVER =
- CLOSED_BOOK =
- OPEN_BOOK =
- GREEN_BOOK =
- BLUE_BOOK =
- ORANGE_BOOK =

- BOOKS =
- NOTEBOOK =
- LEDGER =
- PAGE_WITH_CURL =
- SCROLL =
- PAGE_FACING_UP =
- NEWSPAPER =
- ROLLED_UP_NEWSPAPER =
- BOOKMARK_TABS =
- BOOKMARK =
- LABEL =
- MONEY_BAG =
- COIN =
- YEN_BANKNOTE =
- DOLLAR_BANKNOTE =
- EURO_BANKNOTE =
- POUND_BANKNOTE =
- MONEY_WITH_WINGS =
- CREDIT_CARD =
- RECEIPT =
- CHART_INCREASING_WITH_YEN =
- ENVELOPE =
- E_MAIL =
- INCOMING_ENVELOPE =
- ENVELOPE_WITH_ARROW =
- OUTBOX_TRAY =
- INBOX_TRAY =
- PACKAGE =
- CLOSED_MAILBOX_WITH_RAISED_FLAG =
- CLOSED_MAILBOX_WITH_LOWERED_FLAG =
- OPEN_MAILBOX_WITH_RAISED_FLAG =
- OPEN_MAILBOX_WITH_LOWERED_FLAG =
- POSTBOX =
- BALLOT_BOX_WITH_BALLOT =
- PENCIL =
- BLACK_NIB =

- FOUNTAIN_PEN =
- PEN =
- PAINTBRUSH =
- CRAYON =
- MEMO =
- BRIEFCASE =
- FILE_FOLDER =
- OPEN_FILE_FOLDER =
- CARD_INDEX_DIVIDERS =
- CALENDAR =
- TEAR_OFF_CALENDAR =
- SPIRAL_NOTEPAD =
- SPIRAL_CALENDAR =
- CARD_INDEX =
- CHART_INCREASING =
- CHART_DECREASING =
- BAR_CHART =
- CLIPBOARD =
- PUSHPIN =
- ROUND_PUSHPIN =
- PAPERCLIP =
- LINKED_PAPERCLIPS =
- STRAIGHT_RULER =
- TRIANGULAR_RULER =
- SCISSORS =
- CARD_FILE_BOX =
- FILE_CABINET =
- WASTEBASKET =
- LOCKED =
- UNLOCKED =
- LOCKED_WITH_PEN =
- LOCKED_WITH_KEY =
- KEY =
- OLD_KEY =
- HAMMER =
- AXE =

- PICK =
- HAMMER_AND_PICK =
- HAMMER_AND_WRENCH =
- DAGGER =
- CROSSED_SWORDS =
- PISTOL =
- BOOMERANG =
- BOW_AND_ARROW =
- SHIELD =
- CARPENTRY_SAW =
- WRENCH =
- SCREWDRIVER =
- NUT_AND_BOLT =
- GEAR =
- CLAMP =
- BALANCE_SCALE =
- WHITE_CANE =
- LINK =
- CHAINS =
- HOOK =
- TOOLBOX =
- MAGNET =
- LADDER =
- ALEMBIC =
- TEST_TUBE =
- PETRI_DISH =
- DNA =
- MICROSCOPE =
- TELESCOPE =
- SATELLITE_ANTENNA =
- SYRINGE =
- DROP_OF_BLOOD =
- PILL =
- ADHESIVE_BANDAGE =
- STETHOSCOPE =
- DOOR =

- ELEVATOR =
- MIRROR =
- WINDOW =
- BED =
- COUCH_AND_LAMP =
- CHAIR =
- TOILET =
- PLUNGER =
- SHOWER =
- BATHTUB =
- MOUSE_TRAP =
- RAZOR =
- LOTION_BOTTLE =
- SAFETY_PIN =
- BROOM =
- BASKET =
- ROLL_OF_PAPER =
- BUCKET =
- SOAP =
- TOOTHBRUSH =
- SPONGE =
- FIRE_EXTINGUISHER =
- SHOPPING_CART =
- CIGARETTE =
- COFFIN =
- HEADSTONE =
- FUNERAL_URN =
- MOAI =
- PLACARD =
- ATM_SIGN =
- LITTER_IN_BIN_SIGN =
- POTABLE_WATER =
- WHEELCHAIR_SYMBOL =
- MENS_ROOM =
- WOMENS_ROOM =
- RESTROOM =

- BABY_SYMBOL =
- WATER_CLOSET =
- PASSPORT_CONTROL =
- CUSTOMS =
- BAGGAGE_CLAIM =
- LEFT_LUGGAGE =
- WARNING =
- CHILDREN_CROSSING =
- NO_ENTRY =
- PROHIBITED =
- NO_BICYCLES =
- NO_SMOKING =
- NO_LITTERING =
- NON_POTABLE_WATER =
- NO_PEDESTRIANS =
- NO_MOBILE_PHONES =
- NO_ONE_UNDER_EIGHTEEN =
- RADIOACTIVE =
- BIOHAZARD =
- UP_ARROW =
- UP_RIGHT_ARROW =
- RIGHT_ARROW =
- DOWN_RIGHT_ARROW =
- DOWN_ARROW =
- DOWN_LEFT_ARROW =
- LEFT_ARROW =
- UP_LEFT_ARROW =
- UP_DOWN_ARROW =
- LEFT_RIGHT_ARROW =
- RIGHT_ARROW_CURVING_LEFT =
- LEFT_ARROW_CURVING_RIGHT =
- RIGHT_ARROW_CURVING_UP =
- RIGHT_ARROW_CURVING_DOWN =
- CLOCKWISE_VERTICAL_ARROWS =
- COUNTERCLOCKWISE_ARROWS_BUTTON =
- BACK_ARROW =

- END_ARROW =
- ON_ARROW =
- SOON_ARROW =
- TOP_ARROW =
- PLACE_OF_WORSHIP =
- ATOM_SYMBOL =
- OM =
- STAR_OF_DAVID =
- WHEEL_OF_DHARMA =
- YIN_YANG =
- LATIN_CROSS =
- ORTHODOX_CROSS =
- STAR_AND_CRESCENT =
- PEACE_SYMBOL =
- MENORAH =
- DOTTED_SIX_POINTED_STAR =
- ARIES =
- TAURUS =
- GEMINI =
- CANCER =
- LEO =
- VIRGO =
- LIBRA =
- SCORPIO =
- SAGITTARIUS =
- CAPRICORN =
- AQUARIUS =
- PISCES =
- OPHIUCHUS =
- SHUFFLE_TRACKS_BUTTON =
- REPEAT_BUTTON =
- REPEAT_SINGLE_BUTTON =
- PLAY_BUTTON =
- FAST_FORWARD_BUTTON =
- NEXT_TRACK_BUTTON =
- PLAY_OR_PAUSE_BUTTON =

- REVERSE_BUTTON =
- FAST_REVERSE_BUTTON =
- LAST_TRACK_BUTTON =
- UPWARDS_BUTTON =
- FAST_UP_BUTTON =
- DOWNWARDS_BUTTON =
- FAST_DOWN_BUTTON =
- PAUSE_BUTTON =
- STOP_BUTTON =
- RECORD_BUTTON =
- EJECT_BUTTON =
- CINEMA =
- DIM_BUTTON =
- BRIGHT_BUTTON =
- ANTENNA_BARS =
- VIBRATION_MODE =
- MOBILE_PHONE_OFF =
- FEMALE_SIGN =
- MALE_SIGN =
- TRANSGENDER_SYMBOL =
- MULTIPLY = ×
- PLUS =
- MINUS =
- DIVIDE =
- INFINITY =
- DOUBLE_EXCLAMATION_MARK =
- EXCLAMATION_QUESTION_MARK =
- QUESTION_MARK =
- WHITE_QUESTION_MARK =
- WHITE_EXCLAMATION_MARK =
- EXCLAMATION_MARK =
- WAVY_DASH =
- CURRENCY_EXCHANGE =
- HEAVY_DOLLAR_SIGN =
- MEDICAL_SYMBOL =
- RECYCLING_SYMBOL =

- FLEUR_DE_LIS =
- TRIDENT_EMBLEM =
- NAME_BADGE =
- JAPANESE_SYMBOL_FOR_BEGINNER =
- HOLLOW_RED_CIRCLE =
- CHECK_MARK_BUTTON =
- CHECK_BOX_WITH_CHECK =
- CHECK_MARK = ✓
- CROSS_MARK =
- CROSS_MARK_BUTTON =
- CURLY_LOOP =
- DOUBLE_CURLY_LOOP =
- PART_ALTERNATION_MARK =
- EIGHT_SPOKED_ASTERISK =
- EIGHT_POINTED_STAR =
- SPARKLE =
- COPYRIGHT = ©
- REGISTERED = ®
- TRADE_MARK = ™
- INPUT_LATIN_UPPERCASE =
- INPUT_LATIN_LOWERCASE =
- INPUT_NUMBERS =
- INPUT_SYMBOLS =
- INPUT_LATIN_LETTERS =
- A_BUTTON_BLOOD_TYPE =
- AB_BUTTON_BLOOD_TYPE =
- B_BUTTON_BLOOD_TYPE =
- CL_BUTTON =
- COOL_BUTTON =
- FREE_BUTTON =
- INFORMATION =
- ID_BUTTON =
- CIRCLED_M =
- NEW_BUTTON =
- NG_BUTTON =
- O_BUTTON_BLOOD_TYPE =

- OK_BUTTON =
- P_BUTTON =
- SOS_BUTTON =
- UP_BUTTON =
- VS_BUTTON =
- JAPANESE_HERE_BUTTON =
- JAPANESE_SERVICE_CHARGE_BUTTON =
- JAPANESE_MONTHLY_AMOUNT_BUTTON =
- JAPANESE_NOT_FREE_OF_CHARGE_BUTTON =
- JAPANESE_RESERVED_BUTTON =
- JAPANESE_BARGAIN_BUTTON =
- JAPANESE_DISCOUNT_BUTTON =
- JAPANESE_FREE_OF_CHARGE_BUTTON =
- JAPANESE_PROHIBITED_BUTTON =
- JAPANESE_ACCEPTABLE_BUTTON =
- JAPANESE_APPLICATION_BUTTON =
- JAPANESE_PASSING_GRADE_BUTTON =
- JAPANESE_VACANCY_BUTTON =
- JAPANESE_CONGRATULATIONS_BUTTON =
- JAPANESE_SECRET_BUTTON =
- JAPANESE_OPEN_FOR_BUSINESS_BUTTON =
- JAPANESE_NO_VACANCY_BUTTON =
- RED_CIRCLE =
- ORANGE_CIRCLE =
- YELLOW_CIRCLE =
- GREEN_CIRCLE =
- BLUE_CIRCLE =
- PURPLE_CIRCLE =
- BROWN_CIRCLE =
- BLACK_CIRCLE =
- WHITE_CIRCLE =
- RED_SQUARE =
- ORANGE_SQUARE =
- YELLOW_SQUARE =
- GREEN_SQUARE =
- BLUE_SQUARE =

- PURPLE_SQUARE =
- BROWN_SQUARE =
- BLACK_LARGE_SQUARE =
- WHITE_LARGE_SQUARE =
- BLACK_MEDIUM_SQUARE =
- WHITE_MEDIUM_SQUARE =
- BLACK_MEDIUM_SMALL_SQUARE =
- WHITE_MEDIUM_SMALL_SQUARE =
- BLACK_SMALL_SQUARE =
- WHITE_SMALL_SQUARE =
- LARGE_ORANGE_DIAMOND =
- LARGE_BLUE_DIAMOND =
- SMALL_ORANGE_DIAMOND =
- SMALL_BLUE_DIAMOND =
- RED_TRIANGLE_POINTED_UP =
- RED_TRIANGLE_POINTED_DOWN =
- DIAMOND_WITH_A_DOT =
- RADIO_BUTTON =
- WHITE_SQUARE_BUTTON =
- BLACK_SQUARE_BUTTON =
- CHEQUERED_FLAG =
- TRIANGULAR_FLAG =
- CROSSED_FLAGS =
- BLACK_FLAG =
- WHITE_FLAG =

__init__()

Methods

__init__()	

Attributes

ABACUS
AB_BUTTON_BLOOD_TYPE
ACCORDION
ADHESIVE_BANDAGE
ADMISSION_TICKETS
AERIAL_TRAMWAY
AIRPLANE
AIRPLANE_ARRIVAL
AIRPLANE_DEPARTURE
ALARM_CLOCK
ALEMBIC
ALIEN
ALIEN_MONSTER
AMBULANCE
AMERICAN_FOOTBALL
AMPHORA
ANATOMICAL_HEART
ANCHOR
ANGER_SYMBOL
ANGRY_FACE
ANGRY_FACE_WITH_HORNS
ANGUISHED_FACE
ANT
ANTENNA_BARS
ANXIOUS_FACE_WITH_SWEAT

continues on next page

Table 5 – continued from previous page

AQUARIUS
ARIES
ARTICULATED_LORRY
ARTIST_PALETTE
ASTONISHED_FACE
ATM_SIGN
ATOM_SYMBOL
AUTOMOBILE
AUTO_RICKSHAW
AVOCADO
AXE
A_BUTTON_BLOOD_TYPE
BABY
BABY_ANGEL
BABY_BOTTLE
BABY_CHICK
BABY_SYMBOL
BACKHAND_INDEX_POINTING_DOWN
BACKHAND_INDEX_POINTING_LEFT
BACKHAND_INDEX_POINTING_RIGHT
BACKHAND_INDEX_POINTING_UP
BACKPACK
BACK_ARROW
BACON
BADGER
BADMINTON

continues on next page

Table 5 – continued from previous page

BAGEL
BAGGAGE_CLAIM
BAGUETTE_BREAD
BALANCE_SCALE
BALD
BALL
BALLET_SHOES
BALLOON
BALLOT_BOX_WITH_BALLOT
BANANA
BANJO
BANK
BARBER_POLE
BAR_CHART
BASEBALL
BASKET
BASKETBALL
BAT
BATHTUB
BATTERY
BEACH_WITH_UMBRELLA
BEAMING_FACE_WITH_SMILING_EYES
BEAR
BEATING_HEART
BEAVER
BED

continues on next page

Table 5 – continued from previous page

BEER_MUG
BEETLE
BELL
BELLHOP_BELL
BELL_PEPPER
BELL_WITH_SLASH
BENTO_BOX
BEVERAGE_BOX
BICYCLE
BIKINI
BILLED_CAP
BIOHAZARD
BIRD
BIRTHDAY_CAKE
BISON
BLACK_CIRCLE
BLACK_FLAG
BLACK_HEART
BLACK_LARGE_SQUARE
BLACK_MEDIUM_SMALL_SQUARE
BLACK_MEDIUM_SQUARE
BLACK_NIB
BLACK_SMALL_SQUARE
BLACK_SQUARE_BUTTON
BLOSSOM
BLOWFISH

continues on next page

Table 5 – continued from previous page

BLUEBERRIES
BLUE_BOOK
BLUE_CIRCLE
BLUE_HEART
BLUE_SQUARE
BOAR
BOMB
BONE
BOOKMARK
BOOKMARK_TABS
BOOKS
BOOMERANG
BOTTLE_WITH_POPPING_CORK
BOUQUET
BOWLING
BOWL_WITH_SPOON
BOW_AND_ARROW
BOXING_GLOVE
BOY
BRAIN
BREAD
BREAST_FEEDING
BRICK
BRIDGE_AT_NIGHT
BRIEFCASE
BRIEFS

continues on next page

Table 5 – continued from previous page

BRIGHT_BUTTON
BROCCOLI
BROKEN_HEART
BROOM
BROWN_CIRCLE
BROWN_HEART
BROWN_SQUARE
BUBBLE_TEA
BUCKET
BUG
BUILDING_CONSTRUCTION
BULLET_TRAIN
BURRITO
BUS
BUSTS_IN_SILHOUETTE
BUST_IN_SILHOUETTE
BUS_STOP
BUTTER
BUTTERFLY
B_BUTTON_BLOOD_TYPE
CACTUS
CALENDAR
CALL_ME_HAND
CAMEL
CAMERA
CAMERA_WITH_FLASH

continues on next page

Table 5 – continued from previous page

CAMPING
CANCER
CANDLE
CANDY
CANNED_FOOD
CANOE
CAPRICORN
CARD_FILE_BOX
CARD_INDEX
CARD_INDEX_DIVIDERS
CAROUSEL_HORSE
CARPENTRY_SAW
CARP_STREAMER
CARROT
CASTLE
CAT
CAT_FACE
CAT_WITH_TEARS_OF_JOY
CAT_WITH_WRY_SMILE
CHAINS
CHAIR
CHART_DECREASING
CHART_INCREASING
CHART_INCREASING_WITH_YEN
CHECK_BOX_WITH_CHECK
CHECK_MARK

continues on next page

Table 5 – continued from previous page

CHECK_MARK_BUTTON
CHEESE_WEDGE
CHEQUERED_FLAG
CHERRIES
CHERRY_BLOSSOM
CHESS_PAWN
CHESTNUT
CHICKEN
CHILD
CHILDREN_CROSSING
CHIPMUNK
CHOCOLATE_BAR
CHOPSTICKS
CHRISTMAS_TREE
CHURCH
CIGARETTE
CINEMA
CIRCLED_M
CIRCUS_TENT
CITYSCAPE
CITYSCAPE_AT_DUSK
CLAMP
CLAPPER_BOARD
CLAPPING_HANDS
CLASSICAL_BUILDING
CLINKING_BEER_MUGS

continues on next page

Table 5 – continued from previous page

CLINKING_GLASSES
CLIPBOARD
CLOCKWISE_VERTICAL_ARROWS
CLOSED_BOOK
CLOSED_MAILBOX_WITH_LOWERED_FLAG
CLOSED_MAILBOX_WITH_RAISED_FLAG
CLOSED_UMBRELLA
CLOUD
CLOUD_WITH_LIGHTNING
CLOUD_WITH_LIGHTNING_AND_RAIN
CLOUD_WITH_RAIN
CLOUD_WITH_SNOW
CLOWN_FACE
CLUB_SUIT
CLUTCH_BAG
CL_BUTTON
COAT
COCKROACH
COCKTAIL_GLASS
COCONUT
COFFIN
COIN
COLD_FACE
COLLISION
COMET
COMPASS

continues on next page

Table 5 – continued from previous page

COMPUTER_DISK
COMPUTER_MOUSE
CONFETTI_BALL
CONFOUNDED_FACE
CONFUSED_FACE
CONSTRUCTION
CONSTRUCTION_WORKER
CONTROL_KNOBS
CONVENIENCE_STORE
COOKED_RICE
COOKIE
COOKING
COOL_BUTTON
COPYRIGHT
COUCH_AND_LAMP
COUNTERCLOCKWISE_ARROWS_BUTTON
COUPLE_WITH_HEART
COW
COWBOY_HAT_FACE
COW_FACE
CRAB
CRAYON
CREDIT_CARD
CRESCENT_MOON
CRICKET
CRICKET_GAME

continues on next page

Table 5 – continued from previous page

CROCODILE
CROISSANT
CROSSED_FINGERS
CROSSED_FLAGS
CROSSED_SWORDS
CROSS_MARK
CROSS_MARK_BUTTON
CROWN
CRYING_CAT
CRYING_FACE
CRYSTAL_BALL
CUCUMBER
CUPCAKE
CUP_WITH_STRAW
CURLING_STONE
CURLY_HAIR
CURLY_LOOP
CURRENCY_EXCHANGE
CURRY_RICE
CUSTARD
CUSTOMS
CUT_OF_MEAT
CYCLONE
DAGGER
DANGO
DARK_SKIN_TONE

continues on next page

Table 5 – continued from previous page

DASHING_AWAY
DEAF_PERSON
DECIDUOUS_TREE
DEER
DELIVERY_TRUCK
DEPARTMENT_STORE
DERELICT_HOUSE
DESERT
DESERT_ISLAND
DESKTOP_COMPUTER
DETECTIVE
DIAMOND_SUIT
DIAMOND_WITH_A_DOT
DIM_BUTTON
DIRECT_HIT
DISAPPOINTED_FACE
DISGUISED_FACE
DIVIDE
DIVING_MASK
DIYA_LAMP
DIZZY
DIZZY_FACE
DNA
DODO
DOG
DOG_FACE

continues on next page

Table 5 – continued from previous page

DOLLAR_BANKNOTE
DOLPHIN
DOOR
DOTTED_SIX_POINTED_STAR
DOUBLE_CURLY_LOOP
DOUBLE_EXCLAMATION_MARK
DOUGHNUT
DOVE
DOWNCAST_FACE_WITH_SWEAT
DOWNWARDS_BUTTON
DOWN_ARROW
DOWN_LEFT_ARROW
DOWN_RIGHT_ARROW
DRAGON
DRAGON_FACE
DRESS
DROOLING_FACE
DROPLET
DROP_OF_BLOOD
DRUM
DUCK
DUMPLING
DVD
EAGLE
EAR
EAR_OF_CORN

continues on next page

Table 5 – continued from previous page

EAR_WITH_HEARING_AID
EGG
EGGPLANT
EIGHT_OCLOCK
EIGHT_POINTED_STAR
EIGHT_SPOKED_ASTERISK
EIGHT_THIRTY
EJECT_BUTTON
ELECTRIC_PLUG
ELEPHANT
ELEVATOR
ELEVEN_OCLOCK
ELEVEN_THIRTY
ELF
END_ARROW
ENVELOPE
ENVELOPE_WITH_ARROW
EURO_BANKNOTE
EVERGREEN_TREE
EWE
EXCLAMATION_MARK
EXCLAMATION_QUESTION_MARK
EXPLODING_HEAD
EXPRESSIONLESS_FACE
EYE
EYES

continues on next page

Table 5 – continued from previous page

E_MAIL
FACE_BLOWING_A_KISS
FACE_SAVORING_FOOD
FACE_SCREAMING_IN_FEAR
FACE_VOMITING
FACE_WITHOUT_MOUTH
FACE_WITH_HAND_OVER_MOUTH
FACE_WITH_HEAD_BANDAGE
FACE_WITH_MEDICAL_MASK
FACE_WITH_MONOCLE
FACE_WITH_OPEN_MOUTH
FACE_WITH_RAISED_EYEBROW
FACE_WITH_ROLLING_EYES
FACE_WITH_STEAM_FROM_NOSE
FACE_WITH_SYMBOLS_ON_MOUTH
FACE_WITH_TEARS_OF_JOY
FACE_WITH_THERMOMETER
FACE_WITH_TONGUE
FACTORY
FAIRY
FALAFEL
FALLEN_LEAF
FAMILY
FAST_DOWN_BUTTON
FAST_FORWARD_BUTTON
FAST_REVERSE_BUTTON

continues on next page

Table 5 – continued from previous page

FAST_UP_BUTTON
FAX_MACHINE
FEARFUL_FACE
FEATHER
FEMALE_SIGN
FERRIS_WHEEL
FERRY
FIELD_HOCKEY
FILE_CABINET
FILE_FOLDER
FILM_FRAMES
FILM_PROJECTOR
FIRE
FIRECRACKER
FIREWORKS
FIRE_ENGINE
FIRE_EXTINGUISHER
FIRST_PLACE_MEDAL
FIRST_QUARTER_MOON
FIRST_QUARTER_MOON_FACE
FISH
FISHING_POLE
FISH_CAKE_WITH_SWIRL
FIVE_OCLOCK
FIVE_THIRTY
FLAG_IN_HOLE

continues on next page

Table 5 – continued from previous page

FLAMINGO
FLASHLIGHT
FLATBREAD
FLAT_SHOE
FLEUR_DE_LIS
FLEXED_BICEPS
FLOPPY_DISK
FLOWER_PLAYING_CARDS
FLUSHED_FACE
FLY
FLYING_DISC
FLYING_SAUCER
FOG
FOGGY
FOLDED_HANDS
FONDUE
FOOT
FOOTPRINTS
FORK_AND_KNIFE
FORK_AND_KNIFE_WITH_PLATE
FORTUNE_COOKIE
FOUNTAIN
FOUNTAIN_PEN
FOUR_LEAF_CLOVER
FOUR_OCLOCK
FOUR_THIRTY

continues on next page

Table 5 – continued from previous page

FOX
FRAMED_PICTURE
FREE_BUTTON
FRENCH_FRIES
FRIED_SHRIMP
FROG
FRONT_FACING_BABY_CHICK
FROWNING_FACE
FROWNING_FACE_WITH_OPEN_MOUTH
FUEL_PUMP
FULL_MOON
FULL_MOON_FACE
FUNERAL_URN
GAME_DIE
GARLIC
GEAR
GEMINI
GEM_STONE
GENIE
GHOST
GIRAFFE
GIRL
GLASSES
GLASS_OF_MILK
GLOBE_SHOWING_AMERICAS
GLOBE_SHOWING_ASIA_AUSTRALIA

continues on next page

Table 5 – continued from previous page

GLOBE_SHOWING_EUROPE_AFRICA
GLOBE_WITH_MERIDIANS
GLOVES
GLOWING_STAR
GOAL_NET
GOAT
GOBLIN
GOGGLES
GORILLA
GRADUATION_CAP
GRAPES
GREEN_APPLE
GREEN_BOOK
GREEN_CIRCLE
GREEN_HEART
GREEN_SALAD
GREEN_SQUARE
GRIMACING_FACE
GRINNING_CAT
GRINNING_CAT_WITH_SMILING_EYES
GRINNING_FACE
GRINNING_FACE_WITH_BIG_EYES
GRINNING_FACE_WITH_SMILING_EYES
GRINNING_FACE_WITH_SWEAT
GRINNING_SQUINTING_FACE
GROWING_HEART

continues on next page

Table 5 – continued from previous page

GUARD
GUIDE_DOG
GUITAR
HAMBURGER
HAMMER
HAMMER_AND_PICK
HAMMER_AND_WRENCH
HAMSTER
HANDBAG
HANDSHAKE
HAND_WITH_FINGERS_SPLAYED
HATCHING_CHICK
HEADPHONE
HEADSTONE
HEART_DECORATION
HEART_EXCLAMATION
HEART_SUIT
HEART_WITH_ARROW
HEART_WITH_RIBBON
HEAR_NO_EVIL_MONKEY
HEAVY_DOLLAR_SIGN
HEDGEHOG
HELICOPTER
HERB
HIBISCUS
HIGH_HEELED_SHOE

continues on next page

Table 5 – continued from previous page

HIGH_SPEED_TRAIN
HIGH_VOLTAGE
HIKING_BOOT
HINDU_TEMPLE
HIPPOPOTAMUS
HOLE
HOLLOW_RED_CIRCLE
HONEYBEE
HONEY_POT
HOOK
HORIZONTAL_TRAFFIC_LIGHT
HORSE
HORSE_FACE
HORSE_RACING
HOSPITAL
HOTEL
HOT_BEVERAGE
HOT_DOG
HOT_FACE
HOT_PEPPER
HOT_SPRINGS
HOURGLASS_DONE
HOURGLASS_NOT_DONE
HOUSE
HOUSES
HOUSE_WITH_GARDEN

continues on next page

Table 5 – continued from previous page

HUGGING_FACE
HUNDRED_POINTS
HUSHED_FACE
HUT
ICE
ICE_CREAM
ICE_HOCKEY
ICE_SKATE
ID_BUTTON
INBOX_TRAY
INCOMING_ENVELOPE
INDEX_POINTING_UP
INFINITY
INFORMATION
INPUT_LATIN_LETTERS
INPUT_LATIN_LOWERCASE
INPUT_LATIN_UPPERCASE
INPUT_NUMBERS
INPUT_SYMBOLS
JACK_O_LANTERN
JAPANESE_ACCEPTABLE_BUTTON
JAPANESE_APPLICATION_BUTTON
JAPANESE_BARGAIN_BUTTON
JAPANESE_CASTLE
JAPANESE_CONGRATULATIONS_BUTTON
JAPANESE_DISCOUNT_BUTTON

continues on next page

Table 5 – continued from previous page

JAPANESE_DOLLS
JAPANESE_FREE_OF_CHARGE_BUTTON
JAPANESE_HERE_BUTTON
JAPANESE_MONTHLY_AMOUNT_BUTTON
JAPANESE_NOT_FREE_OF_CHARGE_BUTTON
JAPANESE_NO_VACANCY_BUTTON
JAPANESE_OPEN_FOR_BUSINESS_BUTTON
JAPANESE_PASSING_GRADE_BUTTON
JAPANESE_POST_OFFICE
JAPANESE_PROHIBITED_BUTTON
JAPANESE_RESERVED_BUTTON
JAPANESE_SECRET_BUTTON
JAPANESE_SERVICE_CHARGE_BUTTON
JAPANESE_SYMBOL_FOR_BEGINNER
JAPANESE_VACANCY_BUTTON
JEANS
JOKER
JOYSTICK
KAABA
KANGAROO
KEY
KEYBOARD
KICK_SCOOTER
KIMONO
KISS
KISSING_CAT

continues on next page

Table 5 – continued from previous page

KISSING_FACE
KISSING_FACE_WITH_CLOSED_EYES
KISSING_FACE_WITH_SMILING_EYES
KISS_MARK
KITCHEN_KNIFE
KITE
KIWI_FRUIT
KNOT
KOALA
LABEL
LAB_COAT
LACROSSE
LADDER
LADY_BEETLE
LAPTOP
LARGE_BLUE_DIAMOND
LARGE_ORANGE_DIAMOND
LAST_QUARTER_MOON
LAST_QUARTER_MOON_FACE
LAST_TRACK_BUTTON
LATIN_CROSS
LEAFY_GREEN
LEAF_FLUTTERING_IN_WIND
LEDGER
LEFT_ARROW
LEFT_ARROW_CURVING_RIGHT

continues on next page

Table 5 – continued from previous page

LEFT_FACING_FIST
LEFT_LUGGAGE
LEFT_RIGHT_ARROW
LEFT_SPEECH_BUBBLE
LEG
LEMON
LEO
LEOPARD
LEVEL_SLIDER
LIBRA
LIGHT_BULB
LIGHT_RAIL
LIGHT_SKIN_TONE
LINK
LINKED_PAPERCLIPS
LION
LIPSTICK
LITTER_IN_BIN_SIGN
LIZARD
LLAMA
LOBSTER
LOCKED
LOCKED_WITH_KEY
LOCKED_WITH_PEN
LOCOMOTIVE
LOLLIPOP

continues on next page

Table 5 – continued from previous page

LONG_DRUM
LOTION_BOTTLE
LOUDLY_CRYING_FACE
LOUDSPEAKER
LOVE_HOTEL
LOVE_LETTER
LOVE_YOU_GESTURE
LUGGAGE
LUNGS
LYING_FACE
MAGE
MAGIC_WAND
MAGNET
MAGNIFYING_GLASS_TILTED_LEFT
MAGNIFYING_GLASS_TILTED_RIGHT
MAHJONG_RED_DRAGON
MALE_SIGN
MAMMOTH
MAN
MANGO
MANS_SHOE
MANTELPIECE_CLOCK
MANUAL_WHEELCHAIR
MAN_BEARD
MAN_DANCING
MAPLE_LEAF

continues on next page

Table 5 – continued from previous page

MAP_OF_JAPAN
MARTIAL_ARTS_UNIFORM
MATE
MEAT_ON_BONE
MECHANICAL_ARM
MECHANICAL_LEG
MEDICAL_SYMBOL
MEDIUM_DARK_SKIN_TONE
MEDIUM_LIGHT_SKIN_TONE
MEDIUM_SKIN_TONE
MEGAPHONE
MELON
MEMO
MENORAH
MENS_ROOM
MEN_HOLDING_HANDS
MERPERSON
METRO
MICROBE
MICROPHONE
MICROSCOPE
MIDDLE_FINGER
MILITARY_HELMET
MILITARY_MEDAL
MILKY_WAY
MINIBUS

continues on next page

Table 5 – continued from previous page

MINUS
MIRROR
MOAI
MOBILE_PHONE
MOBILE_PHONE_OFF
MOBILE_PHONE_WITH_ARROW
MONEY_BAG
MONEY_MOUTH_FACE
MONEY_WITH_WINGS
MONKEY
MONKEY_FACE
MONORAIL
MOON_CAKE
MOON_VIEWING_CEREMONY
MOSQUE
MOSQUITO
MOTORCYCLE
MOTORIZED_WHEELCHAIR
MOTORWAY
MOTOR_BOAT
MOTOR_SCOOTER
MOUNTAIN
MOUNTAIN_CABLEWAY
MOUNTAIN_RAILWAY
MOUNT_FUJI
MOUSE

continues on next page

Table 5 – continued from previous page

MOUSE_FACE
MOUSE_TRAP
MOUTH
MOVIE_CAMERA
MRS_CLAUS
MULTIPLY
MUSHROOM
MUSICAL_KEYBOARD
MUSICAL_NOTE
MUSICAL_NOTES
MUSICAL_SCORE
MUTED_SPEAKER
NAIL_POLISH
NAME_BADGE
NATIONAL_PARK
NAUSEATED_FACE
NAZAR_AMULET
NECKTIE
NERD_FACE
NESTING_DOLLS
NEUTRAL_FACE
NEWSPAPER
NEW_BUTTON
NEW_MOON
NEW_MOON_FACE
NEXT_TRACK_BUTTON

continues on next page

Table 5 – continued from previous page

NG_BUTTON
NIGHT_WITH_STARS
NINE_OCLOCK
NINE_THIRTY
NINJA
NON_POTABLE_WATER
NOSE
NOTEBOOK
NOTEBOOK_WITH_DECORATIVE_COVER
NO_BICYCLES
NO_ENTRY
NO_LITTERING
NO_MOBILE_PHONES
NO_ONE_UNDER_EIGHTEEN
NO_PEDESTRIANS
NO_SMOKING
NUT_AND_BOLT
OCTOPUS
ODEN
OFFICE_BUILDING
OGRE
OIL_DRUM
OK_BUTTON
OK_HAND
OLDER_PERSON
OLD_KEY

continues on next page

Table 5 – continued from previous page

OLD_MAN
OLD_WOMAN
OLIVE
OM
ONCOMING_AUTOMOBILE
ONCOMING_BUS
ONCOMING_FIST
ONCOMING_POLICE_CAR
ONCOMING_TAXI
ONE_OCLOCK
ONE_PIECE_SWIMSUIT
ONE_THIRTY
ONION
ON_ARROW
OPEN_BOOK
OPEN_FILE_FOLDER
OPEN_HANDS
OPEN_MAILBOX_WITH_LOWERED_FLAG
OPEN_MAILBOX_WITH_RAISED_FLAG
OPHIUCHUS
OPTICAL_DISK
ORANGE_BOOK
ORANGE_CIRCLE
ORANGE_HEART
ORANGE_SQUARE
ORANGUTAN

continues on next page

Table 5 – continued from previous page

ORTHODOX_CROSS
OTTER
OUTBOX_TRAY
OWL
OX
OYSTER
O_BUTTON_BLOOD_TYPE
PACKAGE
PAGER
PAGE_FACING_UP
PAGE_WITH_CURL
PAINTBRUSH
PALMS_UP_TOGETHER
PALM_TREE
PANCAKES
PANDA
PAPERCLIP
PARACHUTE
PARROT
PARTYING_FACE
PARTY_POPPER
PART_ALTERNATION_MARK
PASSENGER_SHIP
PASSPORT_CONTROL
PAUSE_BUTTON
PAW_PRINTS

continues on next page

Table 5 – continued from previous page

PEACE_SYMBOL
PEACH
PEACOCK
PEANUTS
PEAR
PEN
PENCIL
PENGUIN
PENSIVE_FACE
PEOPLE_HUGGING
PEOPLE_WITH_BUNNY_EARS
PEOPLE_WRESTLING
PERFORMING_ARTS
PERSEVERING_FACE
PERSON
PERSON_BIKING
PERSON_BLOND_HAIR
PERSON_BOUNCING_BALL
PERSON_BOWING
PERSON_CARTWHEELING
PERSON_CLIMBING
PERSON_FACEPALMING
PERSON_FENCING
PERSON_FROWNING
PERSON_GESTURING_NO
PERSON_GESTURING_OK

continues on next page

Table 5 – continued from previous page

PERSON_GETTING_HAIRCUT
PERSON_GETTING_MASSAGE
PERSON_GOLFING
PERSON_IN_BED
PERSON_IN_LOTUS_POSITION
PERSON_IN_STEAMY_ROOM
PERSON_IN_SUIT_LEVITATING
PERSON_IN_TUXEDO
PERSON_JUGGLING
PERSON_KNEELING
PERSON_LIFTING_WEIGHTS
PERSON_MOUNTAIN_BIKING
PERSON_PLAYING_HANDBALL
PERSON_PLAYING_WATER_POLO
PERSON_POUTING
PERSON_RAISING_HAND
PERSON_ROWING_BOAT
PERSON_RUNNING
PERSON_SHRUGGING
PERSON_STANDING
PERSON_SURFING
PERSON_SWIMMING
PERSON_TAKING_BATH
PERSON_TIPPING_HAND
PERSON_WALKING
PERSON_WEARING_TURBAN

continues on next page

Table 5 – continued from previous page

PERSON_WITH_SKULLCAP
PERSON_WITH_VEIL
PETRI_DISH
PICK
PICKUP_TRUCK
PIE
PIG
PIG_FACE
PIG_NOSE
PILE_OF_POO
PILL
PINCHED_FINGERS
PINCHING_HAND
PINEAPPLE
PINE_DECORATION
PING_PONG
PISCES
PISTOL
PIZZA
PIñATA
PLACARD
PLACE_OF_WORSHIP
PLAY_BUTTON
PLAY_OR_PAUSE_BUTTON
PLEADING_FACE
PLUNGER

continues on next page

Table 5 – continued from previous page

PLUS
POLICE_CAR
POLICE_CAR_LIGHT
POLICE_OFFICER
POODLE
POPCORN
POSTAL_HORN
POSTBOX
POST_OFFICE
POTABLE_WATER
POTATO
POTTED_PLANT
POT_OF_FOOD
POULTRY_LEG
POUND_BANKNOTE
POUTING_CAT
POUTING_FACE
PRAYER_BEADS
PREGNANT_WOMAN
PRETZEL
PRINCE
PRINCESS
PRINTER
PROHIBITED
PURPLE_CIRCLE
PURPLE_HEART

continues on next page

Table 5 – continued from previous page

PURPLE_SQUARE
PURSE
PUSHPIN
PUZZLE_PIECE
P_BUTTON
QUESTION_MARK
RABBIT
RABBIT_FACE
RACCOON
RACING_CAR
RADIO
RADIOACTIVE
RADIO_BUTTON
RAILWAY_CAR
RAILWAY_TRACK
RAINBOW
RAISED_BACK_OF_HAND
RAISED_FIST
RAISED_HAND
RAISING_HANDS
RAM
RAT
RAZOR
RECEIPT
RECORD_BUTTON
RECYCLING_SYMBOL

continues on next page

Table 5 – continued from previous page

RED_APPLE
RED_CIRCLE
RED_ENVELOPE
RED_HAIR
RED_HEART
RED_PAPER_LANTERN
RED_SQUARE
RED_TRIANGLE_POINTED_DOWN
RED_TRIANGLE_POINTED_UP
REGISTERED
RELIEVED_FACE
REMINDER_RIBBON
REPEAT_BUTTON
REPEAT_SINGLE_BUTTON
RESCUE_WORKERS_HELMET
RESTROOM
REVERSE_BUTTON
REVOLVING_HEARTS
RHINOCEROS
RIBBON
RICE_BALL
RICE_CRACKER
RIGHT_ANGER_BUBBLE
RIGHT_ARROW
RIGHT_ARROW_CURVING_DOWN
RIGHT_ARROW_CURVING_LEFT

continues on next page

Table 5 – continued from previous page

RIGHT_ARROW_CURVING_UP
RIGHT_FACING_FIST
RING
RINGED_PLANET
ROASTED_SWEET_POTATO
ROBOT
ROCK
ROCKET
ROLLED_UP_NEWSPAPER
ROLLER_COASTER
ROLLER_SKATE
ROLLING_ON_THE_FLOOR_LAUGHING
ROLL_OF_PAPER
ROOSTER
ROSE
ROSETTE
ROUND_PUSHPIN
RUGBY_FOOTBALL
RUNNING_SHIRT
RUNNING_SHOE
SAD_BUT_RELIEVED_FACE
SAFETY_PIN
SAFETY_VEST
SAGITTARIUS
SAILBOAT
SAKE

continues on next page

Table 5 – continued from previous page

SALT
SANDWICH
SANTA_CLAUS
SARI
SATELLITE
SATELLITE_ANTENNA
SAUROPOD
SAXOPHONE
SCARF
SCHOOL
SCISSORS
SCORPIO
SCORPION
SCREWDRIVER
SCROLL
SEAL
SEAT
SECOND_PLACE_MEDAL
SEEDLING
SEE_NO_EVIL_MONKEY
SELFIE
SEVEN_OCLOCK
SEVEN_THIRTY
SEWING_NEEDLE
SHALLOW_PAN_OF_FOOD
SHAMROCK

continues on next page

Table 5 – continued from previous page

SHARK
SHAVED_ICE
SHEAF_OF_RICE
SHIELD
SHINTO_SHRINE
SHIP
SHOOTING_STAR
SHOPPING_BAGS
SHOPPING_CART
SHORTCAKE
SHORTS
SHOWER
SHRIMP
SHUFFLE_TRACKS_BUTTON
SHUSHING_FACE
SIGN_OF_THE_HORNS
SIX_OCLOCK
SIX_THIRTY
SKATEBOARD
SKIER
SKIS
SKULL
SKULL_AND_CROSSBONES
SKUNK
SLED
SLEEPING_FACE

continues on next page

Table 5 – continued from previous page

SLEEPY_FACE
SLIGHTLY_FROWNING_FACE
SLIGHTLY_SMILING_FACE
SLOTH
SLOT_MACHINE
SMALL_AIRPLANE
SMALL_BLUE_DIAMOND
SMALL_ORANGE_DIAMOND
SMILING_CAT_WITH_HEART_EYES
SMILING_FACE
SMILING_FACE_WITH_HALO
SMILING_FACE_WITH_HEARTS
SMILING_FACE_WITH_HEART_EYES
SMILING_FACE_WITH_HORNS
SMILING_FACE_WITH_SMILING_EYES
SMILING_FACE_WITH_SUNGLASSES
SMILING_FACE_WITH_TEAR
SMIRKING_FACE
SNAIL
SNAKE
SNEEZING_FACE
SNOWBOARDER
SNOWFLAKE
SNOWMAN
SNOWMAN_WITHOUT_SNOW
SNOW_CAPPED_MOUNTAIN

continues on next page

Table 5 – continued from previous page

SOAP
SOCCER_BALL
SOCKS
SOFTBALL
SOFT_ICE_CREAM
SOON_ARROW
SOS_BUTTON
SPADE_SUIT
SPAGHETTI
SPARKLE
SPARKLER
SPARKLES
SPARKLING_HEART
SPEAKER_HIGH_VOLUME
SPEAKER_LOW_VOLUME
SPEAKER_MEDIUM_VOLUME
SPEAKING_HEAD
SPEAK_NO_EVIL_MONKEY
SPEECH_BALLOON
SPEEDBOAT
SPIDER
SPIDER_WEB
SPIRAL_CALENDAR
SPIRAL_NOTEPAD
SPIRAL_SHELL
SPONGE

continues on next page

Table 5 – continued from previous page

SPOON
SPORTS_MEDAL
SPORT_UTILITY_VEHICLE
SPOUTING_WHALE
SQUID
SQUINTING_FACE_WITH_TONGUE
STADIUM
STAR
STAR_AND_CRESCENT
STAR_OF_DAVID
STAR_STRUCK
STATION
STATUE_OF_LIBERTY
STEAMING_BOWL
STETHOSCOPE
STOPWATCH
STOP_BUTTON
STOP_SIGN
STRAIGHT_RULER
STRAWBERRY
STUDIO_MICROPHONE
STUFFED_FLATBREAD
SUN
SUNFLOWER
SUNGLASSES
SUNRISE

continues on next page

Table 5 – continued from previous page

SUNRISE_OVER_MOUNTAINS
SUNSET
SUN_BEHIND_CLOUD
SUN_BEHIND_LARGE_CLOUD
SUN_BEHIND_RAIN_CLOUD
SUN_BEHIND_SMALL_CLOUD
SUN_WITH_FACE
SUPERHERO
SUPERVILLAIN
SUSHI
SUSPENSION_RAILWAY
SWAN
SWEAT_DROPLETS
SYNAGOGUE
SYRINGE
TACO
TAKEOUT_BOX
TAMALE
TANABATA_TREE
TANGERINE
TAURUS
TAXI
TEACUP_WITHOUT_HANDLE
TEAPOT
TEAR_OFF_CALENDAR
TEDDY_BEAR

continues on next page

Table 5 – continued from previous page

TELEPHONE
TELEPHONE_RECEIVER
TELESCOPE
TELEVISION
TENNIS
TENT
TEN_OCLOCK
TEN_THIRTY
TEST_TUBE
THERMOMETER
THINKING_FACE
THIRD_PLACE_MEDAL
THONG_SANDAL
THOUGHT_BALLOON
THREAD
THREE_OCLOCK
THREE_THIRTY
THUMBS_DOWN
THUMBS_UP
TICKET
TIGER
TIGER_FACE
TIMER_CLOCK
TIRED_FACE
TOILET
TOKYO_TOWER

continues on next page

Table 5 – continued from previous page

TOMATO
TONGUE
TOOLBOX
TOOTH
TOOTHBRUSH
TOP_ARROW
TOP_HAT
TORNADO
TRACKBALL
TRACTOR
TRADE_MARK
TRAIN
TRAM
TRAM_CAR
TRANSGENDER_SYMBOL
TRIANGULAR_FLAG
TRIANGULAR_RULER
TRIDENT_EMBLEM
TROLLEYBUS
TROPHY
TROPICAL_DRINK
TROPICAL_FISH
TRUMPET
TULIP
TUMBLER_GLASS
TURKEY

continues on next page

Table 5 – continued from previous page

TURTLE
TWELVE_OCLOCK
TWELVE_THIRTY
TWO_HEARTS
TWO_HUMP_CAMEL
TWO_OCLOCK
TWO_THIRTY
T_REX
T_SHIRT
UMBRELLA
UMBRELLA_ON_GROUND
UMBRELLA_WITH_RAIN_DROPS
UNAMUSED_FACE
UNICORN
UNLOCKED
UPSIDE_DOWN_FACE
UPWARDS_BUTTON
UP_ARROW
UP_BUTTON
UP_DOWN_ARROW
UP_LEFT_ARROW
UP_RIGHT_ARROW
VAMPIRE
VERTICAL_TRAFFIC_LIGHT
VIBRATION_MODE
VICTORY_HAND

continues on next page

Table 5 – continued from previous page

VIDEOCASSETTE
VIDEO_CAMERA
VIDEO_GAME
VIOLIN
VIRGO
VOLCANO
VOLLEYBALL
VS_BUTTON
VULCAN_SALUTE
WAFFLE
WANING_CRESCENT_MOON
WANING_GIBBOUS_MOON
WARNING
WASTEBASKET
WATCH
WATERMELON
WATER_BUFFALO
WATER_CLOSET
WATER_WAVE
WAVING_HAND
WAVY_DASH
WAXING_CRESCENT_MOON
WAXING_GIBBOUS_MOON
WEARY_CAT
WEARY_FACE
WEDDING

continues on next page

Table 5 – continued from previous page

WHALE
WHEELCHAIR_SYMBOL
WHEEL_OF_DHARMA
WHITE_CANE
WHITE_CIRCLE
WHITE_EXCLAMATION_MARK
WHITE_FLAG
WHITE_FLOWER
WHITE_HAIR
WHITE_HEART
WHITE_LARGE_SQUARE
WHITE_MEDIUM_SMALL_SQUARE
WHITE_MEDIUM_SQUARE
WHITE_QUESTION_MARK
WHITE_SMALL_SQUARE
WHITE_SQUARE_BUTTON
WILTED_FLOWER
WINDOW
WIND_CHIME
WIND_FACE
WINE_GLASS
WINKING_FACE
WINKING_FACE_WITH_TONGUE
WOLF
WOMAN
WOMANS_BOOT

continues on next page

Table 5 – continued from previous page

WOMANS_CLOTHES
WOMANS_HAT
WOMANS_SANDAL
WOMAN_AND_MAN_HOLDING_HANDS
WOMAN_DANCING
WOMAN_WITH_HEADSCARF
WOMENS_ROOM
WOMEN_HOLDING_HANDS
WOOD
WOOZY_FACE
WORLD_MAP
WORM
WORRIED_FACE
WRAPPED_GIFT
WRENCH
WRITING_HAND
YARN
YAWNING_FACE
YELLOW_CIRCLE
YELLOW_HEART
YELLOW_SQUARE
YEN_BANKNOTE
YIN_YANG
YO_YO
ZANY_FACE
ZEBRA

continues on next page

Table 5 – continued from previous page

ZIPPER_MOUTH_FACE
ZOMBIE
ZZZ

ABACUS = ''

AB_BUTTON_BLOOD_TYPE = ''

ACCORDION = ''

ADHESIVE_BANDAGE = ''

ADMISSION_TICKETS = ''

AERIAL_TRAMWAY = ''

AIRPLANE = ''

AIRPLANE_ARRIVAL = ''

AIRPLANE_DEPARTURE = ''

ALARM_CLOCK = ''

ALEMBIC = ''

ALIEN = ''

ALIEN_MONSTER = ''

AMBULANCE = ''

AMERICAN_FOOTBALL = ''

AMPHORA = ''

ANATOMICAL_HEART = ''

ANCHOR = ''

ANGER_SYMBOL = ''

ANGRY_FACE = ''

ANGRY_FACE_WITH_HORNS = ''

ANGUISHED_FACE = ''

ANT = ''

ANTENNA_BARS = ''

ANXIOUS_FACE_WITH_SWEAT = ''

AQUARIUS = ''

ARIES = ''

ARTICULATED_LORRY = ''

ARTIST_PALETTE = ''

ASTONISHED_FACE = ''

ATM_SIGN = ''

ATOM_SYMBOL = ''

AUTOMOBILE = ''

AUTO_RICKSHAW = ''

AVOCADO = ''

AXE = ''

A_BUTTON_BLOOD_TYPE = ''

BABY = ''

BABY_ANGEL = ''

BABY_BOTTLE = ''

BABY_CHICK = ''

BABY_SYMBOL = ''

BACKHAND_INDEX_POINTING_DOWN = ''

BACKHAND_INDEX_POINTING_LEFT = ''

BACKHAND_INDEX_POINTING_RIGHT = ''

BACKHAND_INDEX_POINTING_UP = ''

BACKPACK = ''

BACK_ARROW = ''

BACON = ''

BADGER = ''

BADMINTON = ''

BAGEL = ''

BAGGAGE_CLAIM = ''

BAGUETTE_BREAD = ''

BALANCE_SCALE = ''

BALD = ''

BALL = ''

BALLET_SHOES = ''

BALLOON = ''

BALLOT_BOX_WITH_BALLOT = ''

BANANA = ''

BANJO = ''

BANK = ''

BARBER_POLE = ''

BAR_CHART = ''

BASEBALL = ''

BASKET = ''

BASKETBALL = ''

BAT = ''

BATHTUB = ''

BATTERY = ''

BEACH_WITH_UMBRELLA = ''

BEAMING_FACE_WITH_SMILING_EYES = ''

BEAR = ''

BEATING_HEART = ''

BEAVER = ''

BED = ''

BEER_MUG = ''

BEETLE = ''

BELL = ''

BELLHOP_BELL = ''

BELL_PEPPER = ''

BELL_WITH_SLASH = ''

BENTO_BOX = ''

BEVERAGE_BOX = ''

BICYCLE = ''

BIKINI = ''

BILLED_CAP = ''

BIOHAZARD = ''

BIRD = ''

BIRTHDAY_CAKE = ''

BISON = ''

BLACK_CIRCLE = ''

BLACK_FLAG = ''

BLACK_HEART = ''

BLACK_LARGE_SQUARE = ''

BLACK_MEDIUM_SMALL_SQUARE = ''

BLACK_MEDIUM_SQUARE = ''

BLACK_NIB = ''

BLACK_SMALL_SQUARE = ''

BLACK_SQUARE_BUTTON = ''

BLOSSOM = ''

BLOWFISH = ''

BLUEBERRIES = ''

BLUE_BOOK = ''

BLUE_CIRCLE = ''

BLUE_HEART = ''

BLUE_SQUARE = ''

BOAR = ''

BOMB = ''

BONE = ''

BOOKMARK = ''

BOOKMARK_TABS = ''

BOOKS = ''

BOOMERANG = ''

BOTTLE_WITH_POPPING_CORK = ''

BOUQUET = ''

BOWLING = ''

BOWL_WITH_SPOON = ''

BOW_AND_ARROW = ''

BOXING_GLOVE = ''

BOY = ''

BRAIN = ''

BREAD = ''

BREAST_FEEDING = ''

BRICK = ''

BRIDGE_AT_NIGHT = ''

BRIEFCASE = ''

BRIEFS = ''

BRIGHT_BUTTON = ''

BROCCOLI = ''

BROKEN_HEART = ''

BROOM = ''

BROWN_CIRCLE = ''

BROWN_HEART = ''

BROWN_SQUARE = ''

BUBBLE_TEA = ''

BUCKET = ''

BUG = ''

BUILDING_CONSTRUCTION = ''

BULLET_TRAIN = ''

BURRITO = ''

BUS = ''

BUSTS_IN_SILHOUETTE = ''

BUST_IN_SILHOUETTE = ''

BUS_STOP = ''

BUTTER = ''

BUTTERFLY = ''

B_BUTTON_BLOOD_TYPE = ''

CACTUS = ''

CALENDAR = ''

CALL_ME_HAND = ''

CAMEL = ''

CAMERA = ''

CAMERA_WITH_FLASH = ''

CAMPING = ''

CANCER = ''

CANDLE = ''

CANDY = ''

CANNED_FOOD = ''

CANOE = ''

CAPRICORN = ''

CARD_FILE_BOX = ''

CARD_INDEX = ''

CARD_INDEX_DIVIDERS = ''

CAROUSEL_HORSE = ''

CARPENTRY_SAW = ''

CARP_STREAMER = ''

CARROT = ''

CASTLE = ''

CAT = ''

CAT_FACE = ''

CAT_WITH_TEARS_OF_JOY = ''

CAT_WITH_WRY_SMILE = ''

CHAINS = ''

CHAIR = ''

CHART_DECREASING = ''

CHART_INCREASING = ''

CHART_INCREASING_WITH_YEN = ''

CHECK_BOX_WITH_CHECK = ''

CHECK_MARK = '✓'

CHECK_MARK_BUTTON = ''

CHEESE_WEDGE = ''

CHEQUERED_FLAG = ''

CHERRIES = ''

CHERRY_BLOSSOM = ''

CHESS_PAWN = ''

CHESTNUT = ''

CHICKEN = ''

CHILD = ''

CHILDREN_CROSSING = ''

CHIPMUNK = ''

CHOCOLATE_BAR = ''

CHOPSTICKS = ''

CHRISTMAS_TREE = ''

CHURCH = ''

CIGARETTE = ''

CINEMA = ''

CIRCLED_M = ''

CIRCUS_TENT = ''

CITYSCAPE = ''

CITYSCAPE_AT_DUSK = ''

CLAMP = ''

CLAPPER_BOARD = ''

CLAPPING_HANDS = ''

CLASSICAL_BUILDING = ''

CLINKING_BEER_MUGS = ''

CLINKING_GLASSES = ''

CLIPBOARD = ''

CLOCKWISE_VERTICAL_ARROWS = ''

CLOSED_BOOK = ''

CLOSED_MAILBOX_WITH_LOWERED_FLAG = ''

CLOSED_MAILBOX_WITH_RAISED_FLAG = ''

CLOSED_UMBRELLA = ''

CLOUD = ''

CLOUD_WITH_LIGHTNING = ''

CLOUD_WITH_LIGHTNING_AND_RAIN = ''

CLOUD_WITH_RAIN = ''

CLOUD_WITH_SNOW = ''

CLOWN_FACE = ''

CLUB_SUIT = ''

CLUTCH_BAG = ''

CL_BUTTON = ''

COAT = ''

COCKROACH = ''

COCKTAIL_GLASS = ''

COCONUT = ''

COFFIN = ''

COIN = ''

COLD_FACE = ''

COLLISION = ''

COMET = ''

COMPASS = ''

COMPUTER_DISK = ''

COMPUTER_MOUSE = ''

CONFETTI_BALL = ''

CONFOUNDED_FACE = ''

CONFUSED_FACE = ''

CONSTRUCTION = ''

CONSTRUCTION_WORKER = ''

CONTROL_KNOBS = ''

CONVENIENCE_STORE = ''

COOKED_RICE = ''

COOKIE = ''

COOKING = ''

COOL_BUTTON = ''

COPYRIGHT = '©'

COUCH_AND_LAMP = ''

COUNTERCLOCKWISE_ARROWS_BUTTON = ''

COUPLE_WITH_HEART = ''

COW = ''

COWBOY_HAT_FACE = ''

COW_FACE = ''

CRAB = ''

CRAYON = ''

CREDIT_CARD = ''

CRESCENT_MOON = ''

CRICKET = ''

CRICKET_GAME = ''

CROCODILE = ''

CROISSANT = ''

CROSSED_FINGERS = ''

CROSSED_FLAGS = ''

CROSSED_SWORDS = ''

CROSS_MARK = ''

CROSS_MARK_BUTTON = ''

CROWN = ''

CRYING_CAT = ''

CRYING_FACE = ''

CRYSTAL_BALL = ''

CUCUMBER = ''

CUPCAKE = ''

CUP_WITH_STRAW = ''

CURLING_STONE = ''

CURLY_HAIR = ''

CURLY_LOOP = ''

CURRENCY_EXCHANGE = ''

CURRY_RICE = ''

CUSTARD = ''

CUSTOMS = ''

CUT_OF_MEAT = ''

CYCLONE = ''

DAGGER = ''

DANGO = ''

DARK_SKIN_TONE = ''

DASHING_AWAY = ''

DEAF_PERSON = ''

DECIDUOUS_TREE = ''

DEER = ''

DELIVERY_TRUCK = ''

DEPARTMENT_STORE = ''

DERELICT_HOUSE = ''

DESERT = ''

DESERT_ISLAND = ''

DESKTOP_COMPUTER = ''

DETECTIVE = ''

DIAMOND_SUIT = ''

DIAMOND_WITH_A_DOT = ''

DIM_BUTTON = ''

DIRECT_HIT = ''

DISAPPOINTED_FACE = ''

DISGUISED_FACE = ''

DIVIDE = ''

DIVING_MASK = ''

DIYA_LAMP = ''

DIZZY = ''

DIZZY_FACE = ''

DNA = ''

DODO = ''

DOG = ''

DOG_FACE = ''

DOLLAR_BANKNOTE = ''

DOLPHIN = ''

DOOR = ''

DOTTED_SIX_POINTED_STAR = ''

DOUBLE_CURLY_LOOP = ''

DOUBLE_EXCLAMATION_MARK = ''

DOUGHNUT = ''

DOVE = ''

DOWNCAST_FACE_WITH_SWEAT = ''

DOWNWARDS_BUTTON = ''

DOWN_ARROW = ''

DOWN_LEFT_ARROW = ''

DOWN_RIGHT_ARROW = ''

DRAGON = ''

DRAGON_FACE = ''

DRESS = ''

DROOLING_FACE = ''

DROPLET = ''

DROP_OF_BLOOD = ''

DRUM = ''

DUCK = ''

DUMPLING = ''

DVD = ''

EAGLE = ''

EAR = ''

EAR_OF_CORN = ''

EAR_WITH_HEARING_AID = ''

EGG = ''

EGGPLANT = ''

EIGHT_OCLOCK = ''

EIGHT_POINTED_STAR = ''

EIGHT_SPOKED_ASTERISK = ''

EIGHT_THIRTY = ''

EJECT_BUTTON = ''

ELECTRIC_PLUG = ''

ELEPHANT = ''

ELEVATOR = ''

ELEVEN_OCLOCK = ''

ELEVEN_THIRTY = ''

ELF = ''

END_ARROW = ''

ENVELOPE = ''

ENVELOPE_WITH_ARROW = ''

EURO_BANKNOTE = ''

EVERGREEN_TREE = ''

EWE = ''

EXCLAMATION_MARK = ''

EXCLAMATION_QUESTION_MARK = ''

EXPLODING_HEAD = ''

EXPRESSIONLESS_FACE = ''

EYE = ''

EYES = ''

E_MAIL = ''

FACE_BLOWING_A_KISS = ''

FACE_SAVORING_FOOD = ''

FACE_SCREAMING_IN_FEAR = ''

FACE_VOMITING = ''

FACE_WITHOUT_MOUTH = ''

FACE_WITH_HAND_OVER_MOUTH = ''

FACE_WITH_HEAD_BANDAGE = ''

FACE_WITH_MEDICAL_MASK = ''

FACE_WITH_MONOCLE = ''

FACE_WITH_OPEN_MOUTH = ''

FACE_WITH_RAISED_EYEBROW = ''

FACE_WITH_ROLLING_EYES = ''

FACE_WITH_STEAM_FROM_NOSE = ''

FACE_WITH_SYMBOLS_ON_MOUTH = ''

FACE_WITH_TEARS_OF_JOY = ''

FACE_WITH_THERMOMETER = ''

FACE_WITH_TONGUE = ''

FACTORY = ''

FAIRY = ''

FALAFEL = ''

FALLEN_LEAF = ''

FAMILY = ''

FAST_DOWN_BUTTON = ''

FAST_FORWARD_BUTTON = ''

FAST_REVERSE_BUTTON = ''

FAST_UP_BUTTON = ''

FAX_MACHINE = ''

FEARFUL_FACE = ''

FEATHER = ''

FEMALE_SIGN = ''

FERRIS_WHEEL = ''

FERRY = ''

FIELD_HOCKEY = ''

FILE_CABINET = ''

FILE_FOLDER = ''

FILM_FRAMES = ''

FILM_PROJECTOR = ''

FIRE = ''

FIRECRACKER = ''

FIREWORKS = ''

FIRE_ENGINE = ''

FIRE_EXTINGUISHER = ''

FIRST_PLACE_MEDAL = ''

FIRST_QUARTER_MOON = ''

FIRST_QUARTER_MOON_FACE = ''

FISH = ''

FISHING_POLE = ''

FISH_CAKE_WITH_SWIRL = ''

FIVE_OCLOCK = ''

FIVE_THIRTY = ''

FLAG_IN_HOLE = ''

FLAMINGO = ''

FLASHLIGHT = ''

FLATBREAD = ''

FLAT_SHOE = ''

FLEUR_DE_LIS = ''

FLEXED_BICEPS = ''

FLOPPY_DISK = ''

FLOWER_PLAYING_CARDS = ''

FLUSHED_FACE = ''

FLY = ''

FLYING_DISC = ''

FLYING_SAUCER = ''

FOG = ''

FOGGY = ''

FOLDED_HANDS = ''

FONDUE = ''

FOOT = ''

FOOTPRINTS = ''

FORK_AND_KNIFE = ''

FORK_AND_KNIFE_WITH_PLATE = ''

FORTUNE_COOKIE = ''

FOUNTAIN = ''

FOUNTAIN_PEN = ''

FOUR_LEAF_CLOVER = ''

FOUR_OCLOCK = ''

FOUR_THIRTY = ''

FOX = ''

FRAMED_PICTURE = ''

FREE_BUTTON = ''

FRENCH_FRIES = ''

FRIED_SHRIMP = ''

FROG = ''

FRONT_FACING_BABY_CHICK = ''

FROWNING_FACE = ''

FROWNING_FACE_WITH_OPEN_MOUTH = ''

FUEL_PUMP = ''

FULL_MOON = ''

FULL_MOON_FACE = ''

FUNERAL_URN = ''

GAME_DIE = ''

GARLIC = ''

GEAR = ''

GEMINI = ''

GEM_STONE = ''

GENIE = ''

GHOST = ''

GIRAFFE = ''

GIRL = ''

GLASSES = ''

GLASS_OF_MILK = ''

GLOBE_SHOWING_AMERICAS = ''

GLOBE_SHOWING_ASIA_AUSTRALIA = ''

GLOBE_SHOWING_EUROPE_AFRICA = ''

GLOBE_WITH_MERIDIANS = ''

GLOVES = ''

GLOWING_STAR = ''

GOAL_NET = ''

GOAT = ''

GOBLIN = ''

GOGGLES = ''

GORILLA = ''

GRADUATION_CAP = ''

GRAPES = ''

GREEN_APPLE = ''

GREEN_BOOK = ''

GREEN_CIRCLE = ''

GREEN_HEART = ''

GREEN_SALAD = ''

GREEN_SQUARE = ''

GRIMACING_FACE = ''

GRINNING_CAT = ''

GRINNING_CAT_WITH_SMILING_EYES = ''

GRINNING_FACE = ''

GRINNING_FACE_WITH_BIG_EYES = ''

GRINNING_FACE_WITH_SMILING_EYES = ''

GRINNING_FACE_WITH_SWEAT = ''

GRINNING_SQUINTING_FACE = ''

GROWING_HEART = ''

GUARD = ''

GUIDE_DOG = ''

GUITAR = ''

HAMBURGER = ''

HAMMER = ''

HAMMER_AND_PICK = ''

HAMMER_AND_WRENCH = ''

HAMSTER = ''

HANDBAG = ''

HANDSHAKE = ''

HAND_WITH_FINGERS_SPLAYED = ''

HATCHING_CHICK = ''

HEADPHONE = ''

HEADSTONE = ''

HEART_DECORATION = ''

HEART_EXCLAMATION = ''

HEART_SUIT = ''

HEART_WITH_ARROW = ''

HEART_WITH_RIBBON = ''

HEAR_NO_EVIL_MONKEY = ''

HEAVY_DOLLAR_SIGN = ''

HEDGEHOG = ''

HELICOPTER = ''

HERB = ''

HIBISCUS = ''

HIGH_HEELED_SHOE = ''

HIGH_SPEED_TRAIN = ''

HIGH_VOLTAGE = ''

HIKING_BOOT = ''

HINDU_TEMPLE = ''

HIPPOPOTAMUS = ''

HOLE = ''

HOLLOW_RED_CIRCLE = ''

HONEYBEE = ''

HONEY_POT = ''

HOOK = ''

HORIZONTAL_TRAFFIC_LIGHT = ''

HORSE = ''

HORSE_FACE = ''

HORSE_RACING = ''

HOSPITAL = ''

HOTEL = ''

HOT_BEVERAGE = ''

HOT_DOG = ''

HOT_FACE = ''

HOT_PEPPER = ''

HOT_SPRINGS = ''

HOURGLASS_DONE = ''

HOURGLASS_NOT_DONE = ''

HOUSE = ''

HOUSES = ''

HOUSE_WITH_GARDEN = ''

HUGGING_FACE = ''

HUNDRED_POINTS = ''

HUSHED_FACE = ''

HUT = ''

ICE = ''

ICE_CREAM = ''

ICE_HOCKEY = ''

ICE_SKATE = ''

ID_BUTTON = ''

INBOX_TRAY = ''

INCOMING_ENVELOPE = ''

INDEX_POINTING_UP = ''

INFINITY = ''

INFORMATION = ''

INPUT_LATIN_LETTERS = ''

INPUT_LATIN_LOWERCASE = ''

INPUT_LATIN_UPPERCASE = ''

INPUT_NUMBERS = ''

INPUT_SYMBOLS = ''

JACK_O_LANTERN = ''

JAPANESE_ACCEPTABLE_BUTTON = ''

JAPANESE_APPLICATION_BUTTON = ''

JAPANESE_BARGAIN_BUTTON = ''

JAPANESE_CASTLE = ''

JAPANESE_CONGRATULATIONS_BUTTON = ''

JAPANESE_DISCOUNT_BUTTON = ''

JAPANESE_DOLLS = ''

JAPANESE_FREE_OF_CHARGE_BUTTON = ''

JAPANESE_HERE_BUTTON = ''

JAPANESE_MONTHLY_AMOUNT_BUTTON = ''

JAPANESE_NOT_FREE_OF_CHARGE_BUTTON = ''

JAPANESE_NO_VACANCY_BUTTON = ''

JAPANESE_OPEN_FOR_BUSINESS_BUTTON = ''

JAPANESE_PASSING_GRADE_BUTTON = ''

JAPANESE_POST_OFFICE = ''

JAPANESE_PROHIBITED_BUTTON = ''

JAPANESE_RESERVED_BUTTON = ''

JAPANESE_SECRET_BUTTON = ''

JAPANESE_SERVICE_CHARGE_BUTTON = ''

JAPANESE_SYMBOL_FOR_BEGINNER = ''

JAPANESE_VACANCY_BUTTON = ''

JEANS = ''

JOKER = ''

JOYSTICK = ''

KAABA = ''

KANGAROO = ''

KEY = ''

KEYBOARD = ''

KICK_SCOOTER = ''

KIMONO = ''

KISS = ''

KISSING_CAT = ''

KISSING_FACE = ''

KISSING_FACE_WITH_CLOSED_EYES = ''

KISSING_FACE_WITH_SMILING_EYES = ''

KISS_MARK = ''

KITCHEN_KNIFE = ''

KITE = ''

KIWI_FRUIT = ''

KNOT = ''

KOALA = ''

LABEL = ''

LAB_COAT = ''

LACROSSE = ''

LADDER = ''

LADY_BEETLE = ''

LAPTOP = ''

LARGE_BLUE_DIAMOND = ''

LARGE_ORANGE_DIAMOND = ''

LAST_QUARTER_MOON = ''

LAST_QUARTER_MOON_FACE = ''

LAST_TRACK_BUTTON = ''

LATIN_CROSS = ''

LEAFY_GREEN = ''

LEAF_FLUTTERING_IN_WIND = ''

LEDGER = ''

LEFT_ARROW = ''

LEFT_ARROW_CURVING_RIGHT = ''

LEFT_FACING_FIST = ''

LEFT_LUGGAGE = ''

LEFT_RIGHT_ARROW = ''

LEFT_SPEECH_BUBBLE = ''

LEG = ''

LEMON = ''

LEO = ''

LEOPARD = ''

LEVEL_SLIDER = ''

LIBRA = ''

LIGHT_BULB = ''

LIGHT_RAIL = ''

LIGHT_SKIN_TONE = ''

LINK = ''

LINKED_PAPERCLIPS = ''

LION = ''

LIPSTICK = ''

LITTER_IN_BIN_SIGN = ''

LIZARD = ''

LLAMA = ''

LOBSTER = ''

LOCKED = ''

LOCKED_WITH_KEY = ''

LOCKED_WITH_PEN = ''

LOCOMOTIVE = ''

LOLLIPOP = ''

LONG_DRUM = ''

LOTION_BOTTLE = ''

LOUDLY_CRYING_FACE = ''

LOUDSPEAKER = ''

LOVE_HOTEL = ''

LOVE_LETTER = ''

LOVE_YOU_GESTURE = ''

LUGGAGE = ''

LUNGS = ''

LYING_FACE = ''

MAGE = ''

MAGIC_WAND = ''

MAGNET = ''

MAGNIFYING_GLASS_TILTED_LEFT = ''

MAGNIFYING_GLASS_TILTED_RIGHT = ''

MAHJONG_RED_DRAGON = ''

MALE_SIGN = ''

MAMMOTH = ''

MAN = ''

MANGO = ''

MANS_SHOE = ''

MANTELPIECE_CLOCK = ''

MANUAL_WHEELCHAIR = ''

MAN_BEARD = ''

MAN_DANCING = ''

MAPLE_LEAF = ''

MAP_OF_JAPAN = ''

MARTIAL_ARTS_UNIFORM = ''

MATE = ''

MEAT_ON_BONE = ''

MECHANICAL_ARM = ''

MECHANICAL_LEG = ''

MEDICAL_SYMBOL = ''

MEDIUM_DARK_SKIN_TONE = ''

MEDIUM_LIGHT_SKIN_TONE = ''

MEDIUM_SKIN_TONE = ''

MEGAPHONE = ''

MELON = ''

MEMO = ''

MENORAH = ''

MENS_ROOM = ''

MEN_HOLDING_HANDS = ''

MERPERSON = ''

METRO = ''

MICROBE = ''

MICROPHONE = ''

MICROSCOPE = ''

MIDDLE_FINGER = ''

MILITARY_HELMET = ''

MILITARY_MEDAL = ''

MILKY_WAY = ''

MINIBUS = ''

MINUS = ''

MIRROR = ''

MOAI = ''

MOBILE_PHONE = ''

MOBILE_PHONE_OFF = ''

MOBILE_PHONE_WITH_ARROW = ''

MONEY_BAG = ''

MONEY_MOUTH_FACE = ''

MONEY_WITH_WINGS = ''

MONKEY = ''

MONKEY_FACE = ''

MONORAIL = ''

MOON_CAKE = ''

MOON_VIEWING_CEREMONY = ''

MOSQUE = ''

MOSQUITO = ''

MOTORCYCLE = ''

MOTORIZED_WHEELCHAIR = ''

MOTORWAY = ''

MOTOR_BOAT = ''

MOTOR_SCOOTER = ''

MOUNTAIN = ''

MOUNTAIN_CABLEWAY = ''

MOUNTAIN_RAILWAY = ''

MOUNT_FUJI = ''

MOUSE = ''

MOUSE_FACE = ''

MOUSE_TRAP = ''

MOUTH = ''

MOVIE_CAMERA = ''

MRS_CLAUS = ''

MULTIPLY = '×'

MUSHROOM = ''

MUSICAL_KEYBOARD = ''

MUSICAL_NOTE = ''

MUSICAL_NOTES = ''

MUSICAL_SCORE = ''

MUTED_SPEAKER = ''

NAIL_POLISH = ''

NAME_BADGE = ''

NATIONAL_PARK = ''

NAUSEATED_FACE = ''

NAZAR_AMULET = ''

NECKTIE = ''

NERD_FACE = ''

NESTING_DOLLS = ''

NEUTRAL_FACE = ''

NEWSPAPER = ''

NEW_BUTTON = ''

NEW_MOON = ''

NEW_MOON_FACE = ''

NEXT_TRACK_BUTTON = ''

NG_BUTTON = ''

NIGHT_WITH_STARS = ''

NINE_OCLOCK = ''

NINE_THIRTY = ''

NINJA = ''

NON_POTABLE_WATER = ''

NOSE = ''

NOTEBOOK = ''

NOTEBOOK_WITH_DECORATIVE_COVER = ''

NO_BICYCLES = ''

NO_ENTRY = ''

NO_LITTERING = ''

NO_MOBILE_PHONES = ''

NO_ONE_UNDER_EIGHTEEN = ''

NO_PEDESTRIANS = ''

NO_SMOKING = ''

NUT_AND_BOLT = ''

OCTOPUS = ''

ODEN = ''

OFFICE_BUILDING = ''

OGRE = ''

OIL_DRUM = ''

OK_BUTTON = ''

OK_HAND = ''

OLDER_PERSON = ''

OLD_KEY = ''

OLD_MAN = ''

OLD_WOMAN = ''

OLIVE = ''

OM = ''

ONCOMING_AUTOMOBILE = ''

ONCOMING_BUS = ''

ONCOMING_FIST = ''

ONCOMING_POLICE_CAR = ''

ONCOMING_TAXI = ''

ONE_OCLOCK = ''

ONE_PIECE_SWIMSUIT = ''

ONE_THIRTY = ''

ONION = ''

ON_ARROW = ''

OPEN_BOOK = ''

OPEN_FILE_FOLDER = ''

OPEN_HANDS = ''

OPEN_MAILBOX_WITH_LOWERED_FLAG = ''

OPEN_MAILBOX_WITH_RAISED_FLAG = ''

OPHIUCHUS = ''

OPTICAL_DISK = ''

ORANGE_BOOK = ''

ORANGE_CIRCLE = ''

ORANGE_HEART = ''

ORANGE_SQUARE = ''

ORANGUTAN = ''

ORTHODOX_CROSS = ''

OTTER = ''

OUTBOX_TRAY = ''

OWL = ''

OX = ''

OYSTER = ''

O_BUTTON_BLOOD_TYPE = ''

PACKAGE = ''

PAGER = ''

PAGE_FACING_UP = ''

PAGE_WITH_CURL = ''

PAINTBRUSH = ''

PALMS_UP_TOGETHER = ''

PALM_TREE = ''

PANCAKES = ''

PANDA = ''

PAPERCLIP = ''

PARACHUTE = ''

PARROT = ''

PARTYING_FACE = ''

PARTY_POPPER = ''

PART_ALTERNATION_MARK = ''

PASSENGER_SHIP = ''

PASSPORT_CONTROL = ''

PAUSE_BUTTON = ''

PAW_PRINTS = ''

PEACE_SYMBOL = ''

PEACH = ''

PEACOCK = ''

PEANUTS = ''

PEAR = ''

PEN = ''

PENCIL = ''

PENGUIN = ''

PENSIVE_FACE = ''

PEOPLE_HUGGING = ''

PEOPLE_WITH_BUNNY_EARS = ''

PEOPLE_WRESTLING = ''

PERFORMING_ARTS = ''

PERSEVERING_FACE = ''

PERSON = ''

PERSON_BIKING = ''

PERSON_BLOND_HAIR = ''

PERSON_BOUNCING_BALL = ''

PERSON_BOWING = ''

PERSON_CARTWHEELING = ''

PERSON_CLIMBING = ''

PERSON_FACEPALMING = ''

PERSON_FENCING = ''

PERSON_FROWNING = ''

PERSON_GESTURING_NO = ''

PERSON_GESTURING_OK = ''

PERSON_GETTING_HAIRCUT = ''

PERSON_GETTING_MASSAGE = ''

PERSON_GOLFING = ''

PERSON_IN_BED = ''

PERSON_IN_LOTUS_POSITION = ''

PERSON_IN_STEAMY_ROOM = ''

PERSON_IN_SUIT_LEVITATING = ''

PERSON_IN_TUXEDO = ''

PERSON_JUGGLING = ''

PERSON_KNEELING = ''

PERSON_LIFTING_WEIGHTS = ''

PERSON_MOUNTAIN_BIKING = ''

PERSON_PLAYING_HANDBALL = ''

PERSON_PLAYING_WATER_POLO = ''

PERSON_POUTING = ''

PERSON_RAISING_HAND = ''

PERSON_ROWING_BOAT = ''

PERSON_RUNNING = ''

PERSON_SHRUGGING = ''

PERSON_STANDING = ''

PERSON_SURFING = ''

PERSON_SWIMMING = ''

PERSON_TAKING_BATH = ''

PERSON_TIPPING_HAND = ''

PERSON_WALKING = ''

PERSON_WEARING_TURBAN = ''

PERSON_WITH_SKULLCAP = ''

PERSON_WITH_VEIL = ''

PETRI_DISH = ''

PICK = ''

PICKUP_TRUCK = ''

PIE = ''

PIG = ''

PIG_FACE = ''

PIG_NOSE = ''

PILE_OF_POO = ''

PILL = ''

PINCHED_FINGERS = ''

PINCHING_HAND = ''

PINEAPPLE = ''

PINE_DECORATION = ''

PING_PONG = ''

PISCES = ''

PISTOL = ''

PIZZA = ''

PIñATA = ''

PLACARD = ''

PLACE_OF_WORSHIP = ''

PLAY_BUTTON = ''

PLAY_OR_PAUSE_BUTTON = ''

PLEADING_FACE = ''

PLUNGER = ''

PLUS = ''

POLICE_CAR = ''

POLICE_CAR_LIGHT = ''

POLICE_OFFICER = ''

POODLE = ''

POPCORN = ''

POSTAL_HORN = ''

POSTBOX = ''

POST_OFFICE = ''

POTABLE_WATER = ''

POTATO = ''

POTTED_PLANT = ''

POT_OF_FOOD = ''

POULTRY_LEG = ''

POUND_BANKNOTE = ''

POUTING_CAT = ''

```
POUTING_FACE = ''

PRAYER_BEADS = ''

PREGNANT_WOMAN = ''

PRETZEL = ''

PRINCE = ''

PRINCESS = ''

PRINTER = ''

PROHIBITED = ''

PURPLE_CIRCLE = ''

PURPLE_HEART = ''

PURPLE_SQUARE = ''

PURSE = ''

PUSHPIN = ''

PUZZLE_PIECE = ''

P_BUTTON = ''

QUESTION_MARK = ''

RABBIT = ''

RABBIT_FACE = ''

RACCOON = ''

RACING_CAR = ''

RADIO = ''

RADIOACTIVE = ''

RADIO_BUTTON = ''

RAILWAY_CAR = ''

RAILWAY_TRACK = ''

RAINBOW = ''

RAISED_BACK_OF_HAND = ''

RAISED_FIST = ''

RAISED_HAND = ''

RAISING_HANDS = ''

RAM = ''
```

RAT = ''

RAZOR = ''

RECEIPT = ''

RECORD_BUTTON = ''

RECYCLING_SYMBOL = ''

RED_APPLE = ''

RED_CIRCLE = ''

RED_ENVELOPE = ''

RED_HAIR = ''

RED_HEART = ''

RED_PAPER_LANTERN = ''

RED_SQUARE = ''

RED_TRIANGLE_POINTED_DOWN = ''

RED_TRIANGLE_POINTED_UP = ''

REGISTERED = '®'

RELIEVED_FACE = ''

REMINDER_RIBBON = ''

REPEAT_BUTTON = ''

REPEAT_SINGLE_BUTTON = ''

RESCUE_WORKERS_HELMET = ''

RESTROOM = ''

REVERSE_BUTTON = ''

REVOLVING_HEARTS = ''

RHINOCEROS = ''

RIBBON = ''

RICE_BALL = ''

RICE_CRACKER = ''

RIGHT_ANGER_BUBBLE = ''

RIGHT_ARROW = ''

RIGHT_ARROW_CURVING_DOWN = ''

RIGHT_ARROW_CURVING_LEFT = ''

RIGHT_ARROW_CURVING_UP = ''

RIGHT_FACING_FIST = ''

RING = ''

RINGED_PLANET = ''

ROASTED_SWEET_POTATO = ''

ROBOT = ''

ROCK = ''

ROCKET = ''

ROLLED_UP_NEWSPAPER = ''

ROLLER_COASTER = ''

ROLLER_SKATE = ''

ROLLING_ON_THE_FLOOR_LAUGHING = ''

ROLL_OF_PAPER = ''

ROOSTER = ''

ROSE = ''

ROSETTE = ''

ROUND_PUSHPIN = ''

RUGBY_FOOTBALL = ''

RUNNING_SHIRT = ''

RUNNING_SHOE = ''

SAD_BUT_RELIEVED_FACE = ''

SAFETY_PIN = ''

SAFETY_VEST = ''

SAGITTARIUS = ''

SAILBOAT = ''

SAKE = ''

SALT = ''

SANDWICH = ''

SANTA_CLAUS = ''

SARI = ''

SATELLITE = ''

SATELLITE_ANTENNA = ''

SAUROPOD = ''

SAXOPHONE = ''

SCARF = ''

SCHOOL = ''

SCISSORS = ''

SCORPIO = ''

SCORPION = ''

SCREWDRIVER = ''

SCROLL = ''

SEAL = ''

SEAT = ''

SECOND_PLACE_MEDAL = ''

SEEDLING = ''

SEE_NO_EVIL_MONKEY = ''

SELFIE = ''

SEVEN_OCLOCK = ''

SEVEN_THIRTY = ''

SEWING_NEEDLE = ''

SHALLOW_PAN_OF_FOOD = ''

SHAMROCK = ''

SHARK = ''

SHAVED_ICE = ''

SHEAF_OF_RICE = ''

SHIELD = ''

SHINTO_SHRINE = ''

SHIP = ''

SHOOTING_STAR = ''

SHOPPING_BAGS = ''

SHOPPING_CART = ''

SHORTCAKE = ''

SHORTS = ''

SHOWER = ''

SHRIMP = ''

SHUFFLE_TRACKS_BUTTON = ''

SHUSHING_FACE = ''

SIGN_OF_THE_HORNS = ''

SIX_OCLOCK = ''

SIX_THIRTY = ''

SKATEBOARD = ''

SKIER = ''

SKIS = ''

SKULL = ''

SKULL_AND_CROSSBONES = ''

SKUNK = ''

SLED = ''

SLEEPING_FACE = ''

SLEEPY_FACE = ''

SLIGHTLY_FROWNING_FACE = ''

SLIGHTLY_SMILING_FACE = ''

SLOTH = ''

SLOT_MACHINE = ''

SMALL_AIRPLANE = ''

SMALL_BLUE_DIAMOND = ''

SMALL_ORANGE_DIAMOND = ''

SMILING_CAT_WITH_HEART_EYES = ''

SMILING_FACE = ''

SMILING_FACE_WITH_HALO = ''

SMILING_FACE_WITH_HEARTS = ''

SMILING_FACE_WITH_HEART_EYES = ''

SMILING_FACE_WITH_HORNS = ''

SMILING_FACE_WITH_SMILING_EYES = ''

SMILING_FACE_WITH_SUNGLASSES = ''

SMILING_FACE_WITH_TEAR = ''

SMIRKING_FACE = ''

SNAIL = ''

SNAKE = ''

SNEEZING_FACE = ''

SNOWBOARDER = ''

SNOWFLAKE = ''

SNOWMAN = ''

SNOWMAN_WITHOUT_SNOW = ''

SNOW_CAPPED_MOUNTAIN = ''

SOAP = ''

SOCCER_BALL = ''

SOCKS = ''

SOFTBALL = ''

SOFT_ICE_CREAM = ''

SOON_ARROW = ''

SOS_BUTTON = ''

SPADE_SUIT = ''

SPAGHETTI = ''

SPARKLE = ''

SPARKLER = ''

SPARKLES = ''

SPARKLING_HEART = ''

SPEAKER_HIGH_VOLUME = ''

SPEAKER_LOW_VOLUME = ''

SPEAKER_MEDIUM_VOLUME = ''

SPEAKING_HEAD = ''

SPEAK_NO_EVIL_MONKEY = ''

SPEECH_BALLOON = ''

SPEEDBOAT = ''

SPIDER = ''

SPIDER_WEB = ''

SPIRAL_CALENDAR = ''

SPIRAL_NOTEPAD = ''

SPIRAL_SHELL = ''

SPONGE = ''

SPOON = ''

SPORTS_MEDAL = ''

SPORT_UTILITY_VEHICLE = ''

SPOUTING_WHALE = ''

SQUID = ''

SQUINTING_FACE_WITH_TONGUE = ''

STADIUM = ''

STAR = ''

STAR_AND_CRESCENT = ''

STAR_OF_DAVID = ''

STAR_STRUCK = ''

STATION = ''

STATUE_OF_LIBERTY = ''

STEAMING_BOWL = ''

STETHOSCOPE = ''

STOPWATCH = ''

STOP_BUTTON = ''

STOP_SIGN = ''

STRAIGHT_RULER = ''

STRAWBERRY = ''

STUDIO_MICROPHONE = ''

STUFFED_FLATBREAD = ''

SUN = ''

SUNFLOWER = ''

SUNGLASSES = ''

SUNRISE = ''

SUNRISE_OVER_MOUNTAINS = ''

SUNSET = ''

SUN_BEHIND_CLOUD = ''

SUN_BEHIND_LARGE_CLOUD = ''

SUN_BEHIND_RAIN_CLOUD = ''

SUN_BEHIND_SMALL_CLOUD = ''

SUN_WITH_FACE = ''

SUPERHERO = ''

SUPERVILLAIN = ''

SUSHI = ''

SUSPENSION_RAILWAY = ''

SWAN = ''

SWEAT_DROPLETS = ''

SYNAGOGUE = ''

SYRINGE = ''

TACO = ''

TAKEOUT_BOX = ''

TAMALE = ''

TANABATA_TREE = ''

TANGERINE = ''

TAURUS = ''

TAXI = ''

TEACUP_WITHOUT_HANDLE = ''

TEAPOT = ''

TEAR_OFF_CALENDAR = ''

TEDDY_BEAR = ''

TELEPHONE = ''

TELEPHONE_RECEIVER = ''

TELESCOPE = ''

TELEVISION = ''

TENNIS = ''

TENT = ''

TEN_OCLOCK = ''

TEN_THIRTY = ''

TEST_TUBE = ''

THERMOMETER = ''

THINKING_FACE = ''

THIRD_PLACE_MEDAL = ''

THONG_SANDAL = ''

THOUGHT_BALLOON = ''

THREAD = ''

THREE_OCLOCK = ''

THREE_THIRTY = ''

THUMBS_DOWN = ''

THUMBS_UP = ''

TICKET = ''

TIGER = ''

TIGER_FACE = ''

TIMER_CLOCK = ''

TIRED_FACE = ''

TOILET = ''

TOKYO_TOWER = ''

TOMATO = ''

TONGUE = ''

TOOLBOX = ''

TOOTH = ''

TOOTHBRUSH = ''

TOP_ARROW = ''

TOP_HAT = ''

TORNADO = ''

TRACKBALL = ''

TRACTOR = ''

TRADE_MARK = '™'

TRAIN = ''

TRAM = ''

TRAM_CAR = ''

TRANSGENDER_SYMBOL = ''

TRIANGULAR_FLAG = ''

TRIANGULAR_RULER = ''

TRIDENT_EMBLEM = ''

TROLLEYBUS = ''

TROPHY = ''

TROPICAL_DRINK = ''

TROPICAL_FISH = ''

TRUMPET = ''

TULIP = ''

TUMBLER_GLASS = ''

TURKEY = ''

TURTLE = ''

TWELVE_OCLOCK = ''

TWELVE_THIRTY = ''

TWO_HEARTS = ''

TWO_HUMP_CAMEL = ''

TWO_OCLOCK = ''

TWO_THIRTY = ''

T_REX = ''

T_SHIRT = ''

UMBRELLA = ''

UMBRELLA_ON_GROUND = ''

UMBRELLA_WITH_RAIN_DROPS = ''

UNAMUSED_FACE = ''

UNICORN = ''

UNLOCKED = ''

UPSIDE_DOWN_FACE = ''

UPWARDS_BUTTON = ''

UP_ARROW = ''

UP_BUTTON = ''

UP_DOWN_ARROW = ''

UP_LEFT_ARROW = ''

UP_RIGHT_ARROW = ''

VAMPIRE = ''

VERTICAL_TRAFFIC_LIGHT = ''

VIBRATION_MODE = ''

VICTORY_HAND = ''

VIDEOCASSETTE = ''

VIDEO_CAMERA = ''

VIDEO_GAME = ''

VIOLIN = ''

VIRGO = ''

VOLCANO = ''

VOLLEYBALL = ''

VS_BUTTON = ''

VULCAN_SALUTE = ''

WAFFLE = ''

WANING_CRESCENT_MOON = ''

WANING_GIBBOUS_MOON = ''

WARNING = ''

WASTEBASKET = ''

WATCH = ''

WATERMELON = ''

WATER_BUFFALO = ''

WATER_CLOSET = ''

WATER_WAVE = ''

WAVING_HAND = ''

WAVY_DASH = ''

WAXING_CRESCENT_MOON = ''

WAXING_GIBBOUS_MOON = ''

WEARY_CAT = ''

WEARY_FACE = ''

WEDDING = ''

WHALE = ''

WHEELCHAIR_SYMBOL = ''

WHEEL_OF_DHARMA = ''

WHITE_CANE = ''

WHITE_CIRCLE = ''

WHITE_EXCLAMATION_MARK = ''

WHITE_FLAG = ''

WHITE_FLOWER = ''

WHITE_HAIR = ''

WHITE_HEART = ''

WHITE_LARGE_SQUARE = ''

WHITE_MEDIUM_SMALL_SQUARE = ''

WHITE_MEDIUM_SQUARE = ''

WHITE_QUESTION_MARK = ''

WHITE_SMALL_SQUARE = ''

WHITE_SQUARE_BUTTON = ''

WILTED_FLOWER = ''

WINDOW = ''

WIND_CHIME = ''

WIND_FACE = ''

WINE_GLASS = ''

WINKING_FACE = ''

WINKING_FACE_WITH_TONGUE = ''

WOLF = ''

WOMAN = ''

WOMANS_BOOT = ''

WOMANS_CLOTHES = ''

WOMANS_HAT = ''

WOMANS_SANDAL = ''

WOMAN_AND_MAN_HOLDING_HANDS = ''

WOMAN_DANCING = ''

WOMAN_WITH_HEADSCARF = ''

WOMENS_ROOM = ''

WOMEN_HOLDING_HANDS = ''

WOOD = ''

WOOZY_FACE = ''

WORLD_MAP = ''

WORM = ''

WORRIED_FACE = ''

WRAPPED_GIFT = ''

WRENCH = ''

WRITING_HAND = ''

YARN = ''

YAWNING_FACE = ''

YELLOW_CIRCLE = ''

YELLOW_HEART = ''

YELLOW_SQUARE = ''

YEN_BANKNOTE = ''

YIN_YANG = ''

YO_YO = ''

ZANY_FACE = ''

ZEBRA = ''

ZIPPER_MOUTH_FACE = ''

ZOMBIE = ''

ZZZ = ''

3.2.2 Fonts

Fonts in the pygamelib are nothing more than a specially organized sprite collection.

The way to use it is extremely simple: you instantiate a Font object and ask it to load the data from a specific font.

For example to load the 8bits font, you do:

Example::

> from pygamelib.gfx import core
>
> my_font = core.Font('8bits')

That's it! The you can use it to format Text objects.

8bits

New in version 1.3.0.

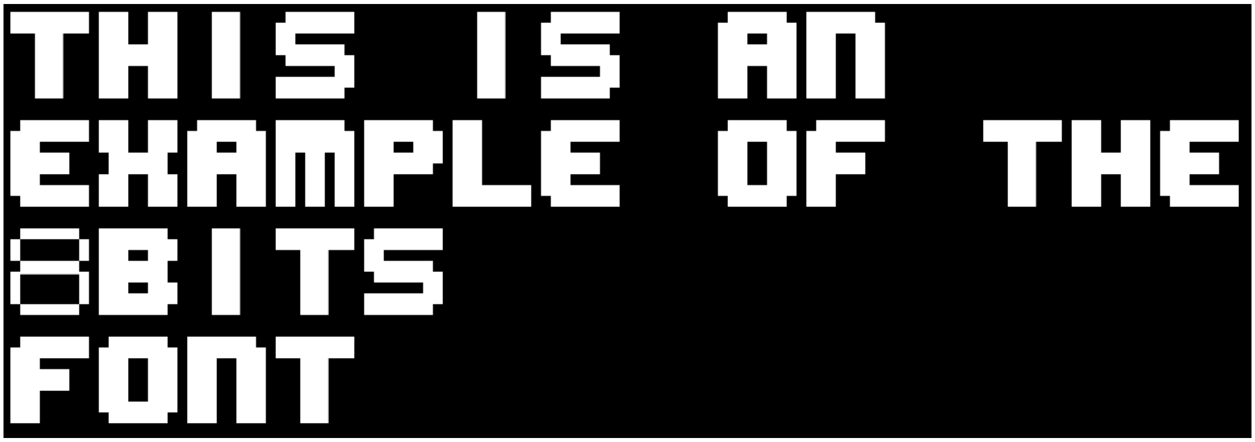
How to use?

Example:

```
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("8bits")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



THIS IS AN
EXAMPLE OF THE
8BITS
FONT

figlet-caligraphy

New in version 1.3.0.

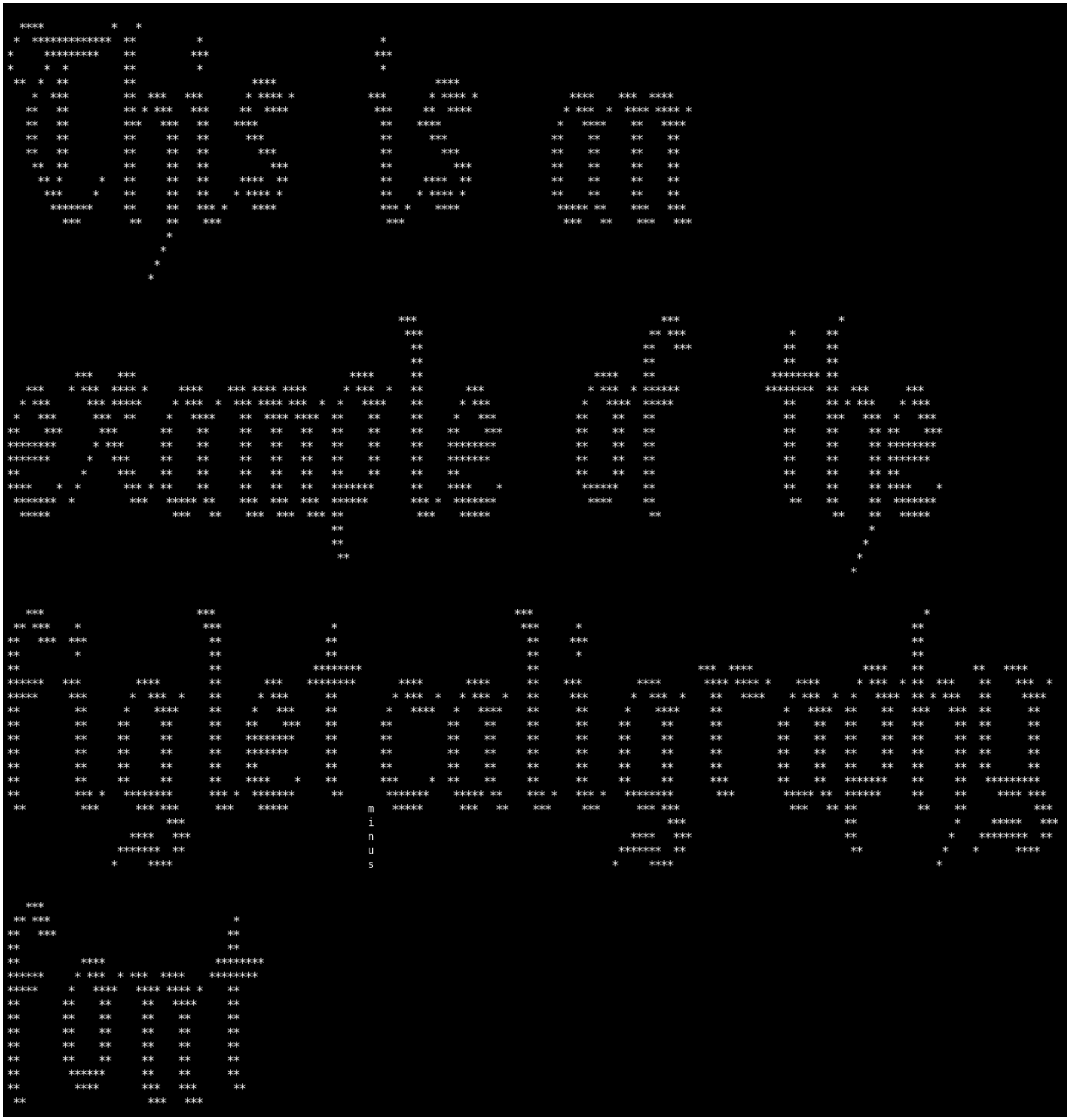
How to use?

Example:

```
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-caligraphy")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by “figlet-” are but a small portion of what’s available here.

To easily convert FIGlet fonts to the pygamelib’s format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-doom

New in version 1.3.0.

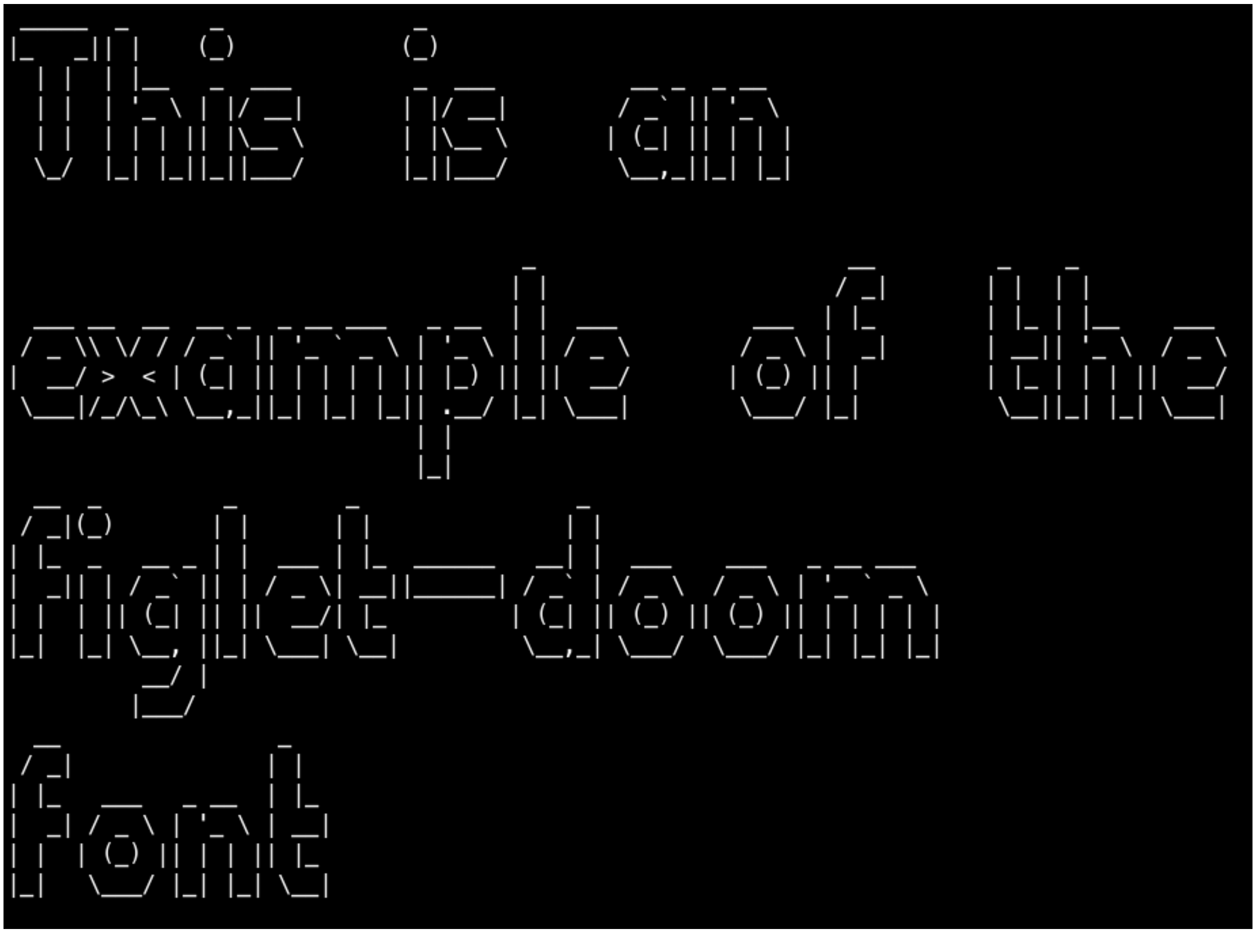
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-doom")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
 _____  _      _          _
|_   _|| |    (_)        (_)
  | |  | |__   _  ___     _  ___      __ _  _ __
  | |  | '_ \ | |/ __|   | |/ __|    / _` || '_ \
  | |  | | | || |\__ \   | |\__ \   | (_| || | | |
  \_/  |_| |_||_||___/   |_||___/    \__,_||_| |_|


                                      _                   __     _    _
                                     | |                 / _|   | |  | |
  ___ __  __  __ _  _ __ ___   _ __  | |  ___      ___  | |_    | |_ | |__    ___
 / _ \\ \/ / / _` || '_ ` _ \ | '_ \ | | / _ \    / _ \ |  _|   | __|| '_ \  / _ \
|  __/ >  < | (_| || | | | | || |_) || ||  __/   | (_) || |     | |_ | | | ||  __/
 \___|/_/\_\ \__,_||_| |_| |_|| .__/ |_| \___|    \___/ |_|      \__||_| |_| \___|
                              | |
                              |_|
  __  _         _        _                _
 / _|(_)       | |      | |              | |
| |_  _   __ _ | |  ___ | |_  ______   __| |  ___    ___   _ __ ___
|  _|| | / _` || | / _ \| __||______| / _` | / _ \  / _ \ | '_ ` _ \
| |  | || (_| || ||  __/| |_         | (_| || (_) || (_) || | | | | |
|_|  |_| \__, ||_| \___| \__|         \__,_| \___/  \___/ |_| |_| |_|
          __/ |
         |___/
  __                _
 / _|              | |
| |_   ___   _ __  | |_
|  _| / _ \ | '_ \ | __|
| |  | (_) || | | || |_
|_|   \___/ |_| |_| \__|
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-graffiti

New in version 1.3.0.

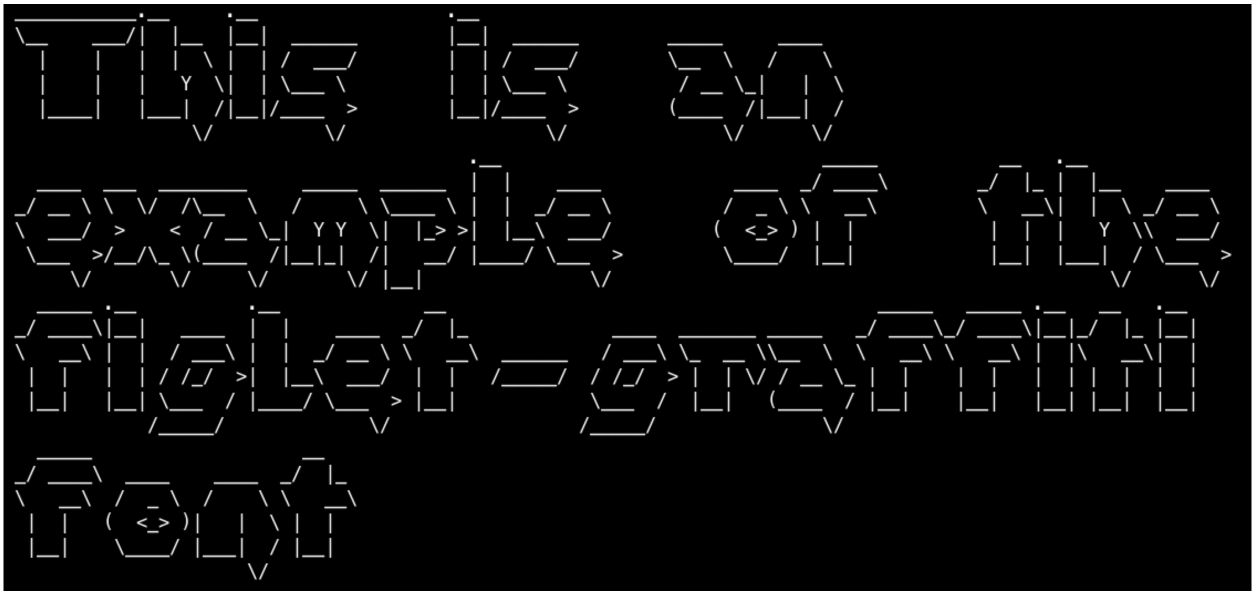
How to use?

Example:

```
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-graffiti")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
___________.__     .__                 .__
\__    ___/|  |__  |__|  ______        |__|  ______        _____     ____
  |    |   |  |  \ |  | /  ___/        |  | /  ___/        \__  \   /    \
  |    |   |   Y  \|  | \___ \         |  | \___ \          / __ \_|   |  \
  |____|   |___|  /|__|/____  >        |__|/____  >        (____  /|___|  /
                \/          \/                  \/              \/      \/
                                         .__                             _____               __    .__
  ____  ___  ________     _____  ______  |  |    ____            ____  _/ ____\        _/  |_ |  |__    ____
_/ __ \ \  \/  /\__  \   /     \ \____ \ |  |  _/ __ \          /  _ \ \   __\         \   __\|  |  \ _/ __ \
\  ___/  >    <  / __ \_|  Y Y  \|  |_> >|  |__\  ___/         (  <_> ) |  |            |  |  |   Y  \\  ___/
 \___  >/__/\_ \(____  /|__|_|  /|   __/ |____/ \___  >         \____/  |__|            |__|  |___|  / \___  >
     \/       \/     \/       \/ |__|               \/                                             \/      \/
  _____ .__         .__             __                                       _____   _____ .__   __   .__
_/ ____\|__|  ____  |  |    ____  _/  |_             ____  _______ _____   _/ ____\_/ ____\|__|_/  |_ |__|
\   __\ |  | / ___\ |  |  _/ __ \ \   __\  ______   / ___\ \_  __ \\__  \  \   __\ \   __\ |  |\   __\|  |
 |  |   |  |/ /_/  >|  |__\  ___/  |  |   /_____/  / /_/  > |  | \/ / __ \_ |  |    |  |   |  | |  |  |  |
 |__|   |__|\___  / |____/ \___  > |__|            \___  /  |__|   (____  / |__|    |__|   |__| |__|  |__|
            /_____/            \/                  /_____/              \/
  _____                   __
_/ ____\  ____    ____  _/  |_
\   __\  /  _ \  /    \ \   __\
 |  |   (  <_> )|   |  \ |  |
 |__|    \____/ |___|  / |__|
                     \/
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-mirror

New in version 1.3.0.

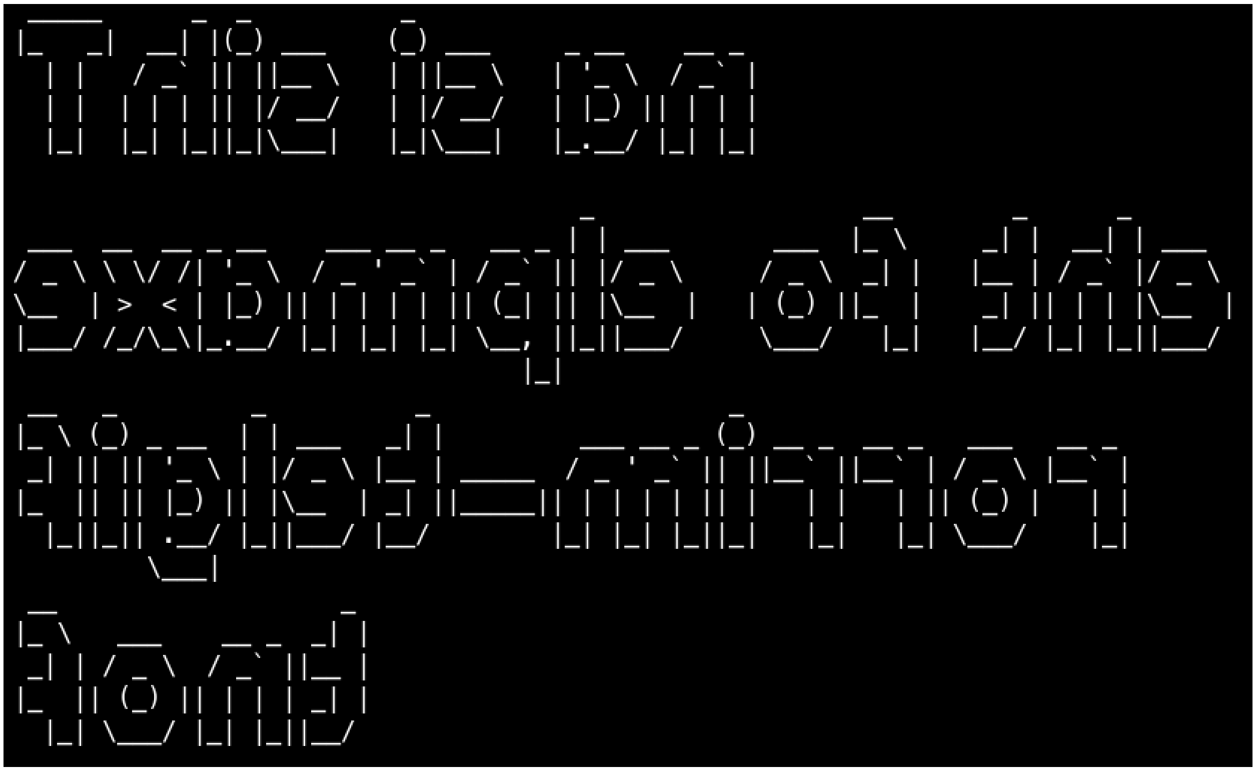
How to use?

Example:

```
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-mirror")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
 _____      _  _          _
|_   _|  __| |(_) ___    (_) ___     _ __    __ _
  | |   / _` || ||__ \   | ||__ \   | '_ \  / _` |
  | |  | | | | || |/ __/   | |/ __/   | |_) || | | |
  |_|  |_| |_||_|\___|   |_|\___|   |_.__/ |_| |_|

                                      _                  __        _      _
 ___  __  __ _ __    ___ __ _   __ _ | | ___       ___  |_ \     _| |  __| | ___
/ _ \ \ \/ /| '_ \  / _ ' _` | / _` || ||/ _ \     / _ \  _| |   |__ | / _` ||/ _ \
\__  | >  < | |_) || | | | | || (_| || ||\__  |   | (_) ||_  |    _| || | | | ||\__  |
|___/ /_/\_\|_.__/ |_| |_| |_| \__, ||_||___/     \___/   |_|   |__/ |_| |_||___/
                                  |_|
 __   _         _          _                    _
|_ \ (_) _ __  | | ___   _| |         ___ __ _ (_) __ _  __ _   ___   __ _
 _| || || '_ \ | |/ _ \ |__ | _____  / _ ' _` || ||__` ||__` | / _ \ |__` |
|_  || || |_) || |\__  | _| ||_____|| | | | | || |   | |   | || (_) |   | |
  |_||_|| .__/ |_||___/ |__/        |_| |_| |_||_|   |_|   |_| \___/    |_|
         \___|
 __                   _
|_ \   ___    __ _  _| |
 _| | / _ \  / _` ||__ |
|_  || (_) || | | | _| |
  |_| \___/ |_| |_||__/
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-pepper

New in version 1.3.0.

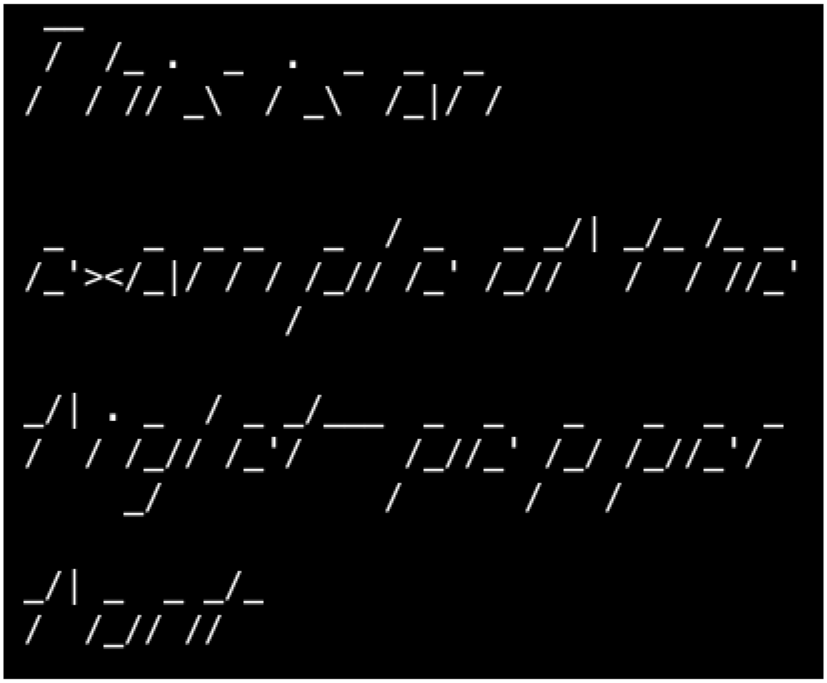
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-pepper")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
 __
 /  /_ .  _  .  _  _  _
/  / // _\  / _\  /_|/ /


 _    _  _ _   _  / _   _ _/| _/_ /_ _
/_'></_|/ / / /_// /_' /_//   /  / //_'
             /

_/| . _  / _ _/___  _  _   _   _  _  _
/  / /_// /_'/     /_//_' /_/ /_//_'/
     _/           /      /   /

_/| _  _ _/_
/  /_// //
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by “figlet-” are but a small portion of what’s available here.

To easily convert FIGlet fonts to the pygamelib’s format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-poison

New in version 1.3.0.

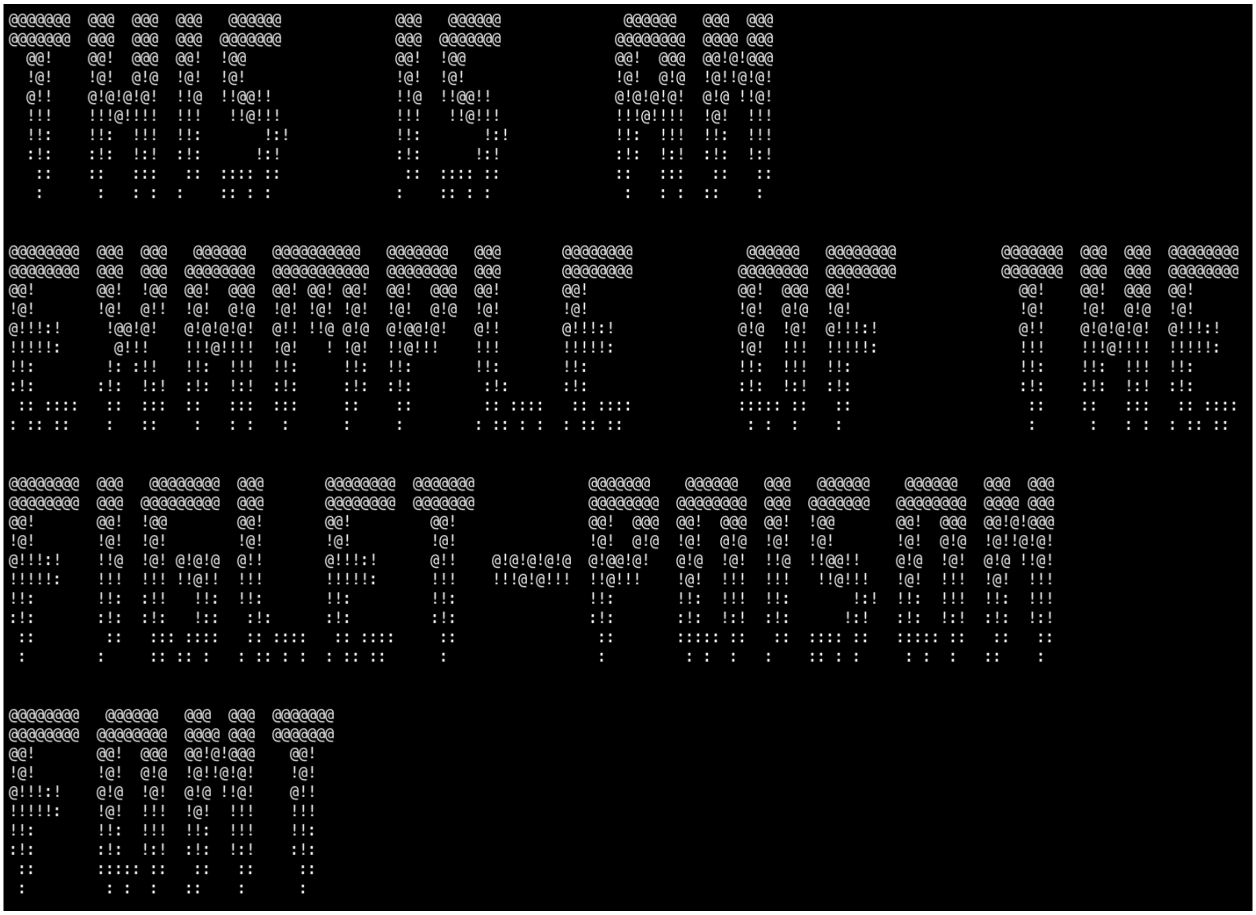
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-poison")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
@@@@@@@  @@@  @@@  @@@   @@@@@@             @@@   @@@@@@              @@@@@@   @@@  @@@
@@@@@@@  @@@  @@@  @@@  @@@@@@@             @@@  @@@@@@@             @@@@@@@@  @@@@ @@@
  @@!    @@!  @@@  @@!  !@@                 @@!  !@@                 @@!  @@@  @@!@!@@@
  !@!    !@!  @!@  !@!  !@!                 !@!  !@!                 !@!  @!@  !@!!@!@!
  @!!    @!@!@!@!  !!@  !!@@!!              !!@  !!@@!!              @!@!@!@!  @!@ !!@!
  !!!    !!!@!!!!  !!!   !!@!!!             !!!   !!@!!!             !!!@!!!!  !@!  !!!
  !!:    !!:  !!!  !!:       !:!            !!:       !:!            !!:  !!!  !!:  !!!
  :!:    :!:  !:!  :!:      !:!             :!:      !:!             :!:  !:!  :!:  !:!
   ::    ::   :::   ::  :::: ::              ::  :::: ::             ::   :::   ::   ::
   :      :   : :  :    :: : :              :    :: : :               :   : :  ::    :

@@@@@@@@  @@@  @@@   @@@@@@   @@@@@@@@@@   @@@@@@@   @@@       @@@@@@@@             @@@@@@   @@@@@@@@            @@@@@@@  @@@  @@@  @@@@@@@@
@@@@@@@@  @@@  @@@  @@@@@@@@  @@@@@@@@@@@  @@@@@@@@  @@@       @@@@@@@@            @@@@@@@@  @@@@@@@@            @@@@@@@  @@@  @@@  @@@@@@@@
@@!       @@!  !@@  @@!  @@@  @@! @@! @@!  @@!  @@@  @@!       @@!                 @@!  @@@  @@!                   @@!    @@!  @@@  @@!
!@!       !@!  @!!  !@!  @!@  !@! !@! !@!  !@!  @!@  !@!       !@!                 !@!  @!@  !@!                   !@!    !@!  @!@  !@!
@!!!:!     !@@!@!   @!@!@!@!  @!! !!@ @!@  @!@@!@!   @!!       @!!!:!              @!@  !@!  @!!!:!                @!!    @!@!@!@!  @!!!:!
!!!!!:      @!!!    !!!@!!!!  !@!   ! !@!  !!@!!!    !!!       !!!!!:              !@!  !!!  !!!!!:                !!!    !!!@!!!!  !!!!!:
!!:        !: :!!   !!:  !!!  !!:     !!:  !!:       !!:       !!:                 !!:  !!!  !!:                   !!:    !!:  !!!  !!:
:!:       :!:  !:!  :!:  !:!  :!:     :!:  :!:        :!:      :!:                 :!:  !:!  :!:                   :!:    :!:  !:!  :!:
 :: ::::   ::  :::  ::   :::  :::     ::    ::        :: ::::   :: ::::            ::::: ::   ::                    ::    ::   :::   :: ::::
: :: ::    :   ::    :   : :   :      :     :        : :: : :  : :: ::              : :  :    :                     :      :   : :  : :: ::

@@@@@@@@  @@@   @@@@@@@@  @@@       @@@@@@@@  @@@@@@@             @@@@@@@    @@@@@@   @@@   @@@@@@    @@@@@@   @@@  @@@
@@@@@@@@  @@@  @@@@@@@@@  @@@       @@@@@@@@  @@@@@@@             @@@@@@@@  @@@@@@@@  @@@  @@@@@@@   @@@@@@@@  @@@@ @@@
@@!       @@!  !@@        @@!       @@!         @@!               @@!  @@@  @@!  @@@  @@!  !@@       @@!  @@@  @@!@!@@@
!@!       !@!  !@!        !@!       !@!         !@!               !@!  @!@  !@!  @!@  !@!  !@!       !@!  @!@  !@!!@!@!
@!!!:!    !!@  !@! @!@!@  @!!       @!!!:!      @!!    @!@!@!@!@  @!@@!@!   @!@  !@!  !!@  !!@@!!    @!@  !@!  @!@ !!@!
!!!!!:    !!!  !!! !!@!!  !!!       !!!!!:      !!!    !!!@!@!!!  !!@!!!    !@!  !!!  !!!   !!@!!!   !@!  !!!  !@!  !!!
!!:       !!:  :!!   !!:  !!:       !!:         !!:               !!:       !!:  !!!  !!:       !:!  !!:  !!!  !!:  !!!
:!:       :!:  :!:   !::   :!:      :!:         :!:               :!:       :!:  !:!  :!:      !:!   :!:  !:!  :!:  !:!
 ::        ::   ::: ::::   :: ::::   :: ::::     ::                ::       ::::: ::   ::  :::: ::   ::::: ::   ::   ::
 :        :     :: :: :   : :: : :  : :: ::      :                 :         : :  :   :    :: : :     : :  :   ::    :

@@@@@@@@   @@@@@@   @@@  @@@  @@@@@@@
@@@@@@@@  @@@@@@@@  @@@@ @@@  @@@@@@@
@@!       @@!  @@@  @@!@!@@@    @@!
!@!       !@!  @!@  !@!!@!@!    !@!
@!!!:!    @!@  !@!  @!@ !!@!    @!!
!!!!!:    !@!  !!!  !@!  !!!    !!!
!!:       !!:  !!!  !!:  !!!    !!:
:!:       :!:  !:!  :!:  !:!    :!:
 ::       ::::: ::   ::   ::     ::
 :         : :  :   ::    :      :
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by “figlet-” are but a small portion of what’s available here.

To easily convert FIGlet fonts to the pygamelib’s format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-puffy

New in version 1.3.0.

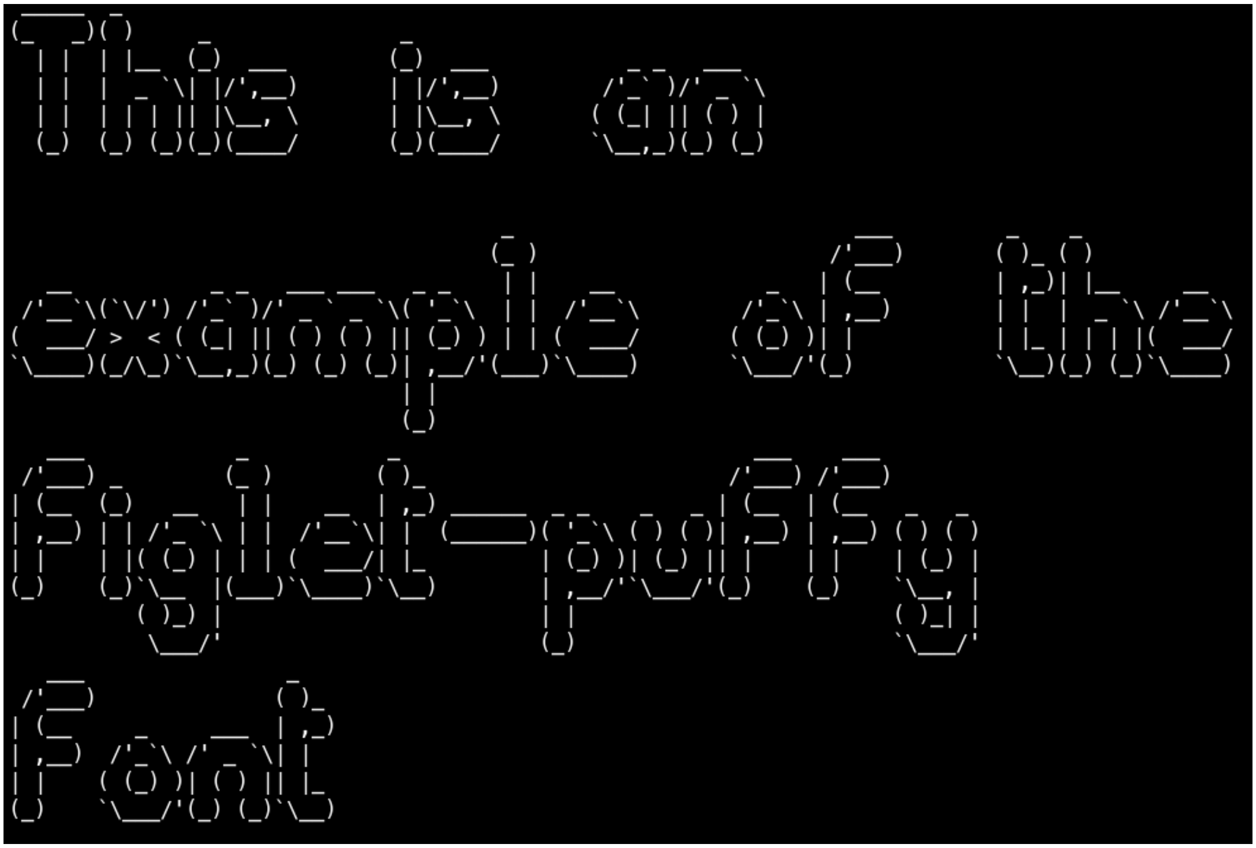
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-puffy")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
 _____  _
(_   _)( )     _               _
  | |  | |__  (_)  ___        (_)  ___           _ _   ___
  | |  |  _ `\| |/',__)       | |/',__)        /'_` )/' _ `\
  | |  | | | || |\__, \       | |\__, \       ( (_| || ( ) |
  (_)  (_) (_)(_)(____/       (_)(____/       `\__,_)(_) (_)

                                       _                           ___         _    _
                                      (_ )                       /'___)       ( )_ ( )
   __           _ _   ___ ___   _ _    | |    __            _   | (__         | ,_)| |__     __
 /'__`\(`\/') /'_` )/' _ ` _ `\( '_`\  | |  /'__`\        /'_`\ | ,__)        | |  |  _ `\ /'__`\
(  ___/ >  < ( (_| || ( ) ( ) || (_) ) | | (  ___/       ( (_) )| |           | |_ | | | |(  ___/
`\____)(_/\_)`\__,_)(_) (_) (_)| ,__/'(___)`\____)       `\___/'(_)           `\__)(_) (_)`\____)
                               | |
                               (_)

   ___            _           _                           ___    ___
 /'___) _        (_ )        ( )_                       /'___) /'___)
| (__  (_)   __   | |    __  | ,_) _____  _ _    _   _ | (__  | (__   _   _
| ,__) | | /'_ `\ | |  /'__`\| |  (_____)( '_`\ ( ) ( )| ,__) | ,__) ( ) ( )
| |    | |( (_) | | | (  ___/| |_        | (_) )| (_) || |    | |    | (_) |
(_)    (_)`\__  |(___)`\____)`\__)       | ,__/'`\___/'(_)    (_)    `\__, |
          ( )_) |                        | |                         ( )_| |
           \___/'                        (_)                         `\___/'

   ___                _
 /'___)              ( )_
| (__     _     ___  | ,_)
| ,__)  /'_`\ /' _ `\| |
| |    ( (_) )| ( ) || |_
(_)    `\___/'(_) (_)`\__)
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-rounded

New in version 1.3.0.

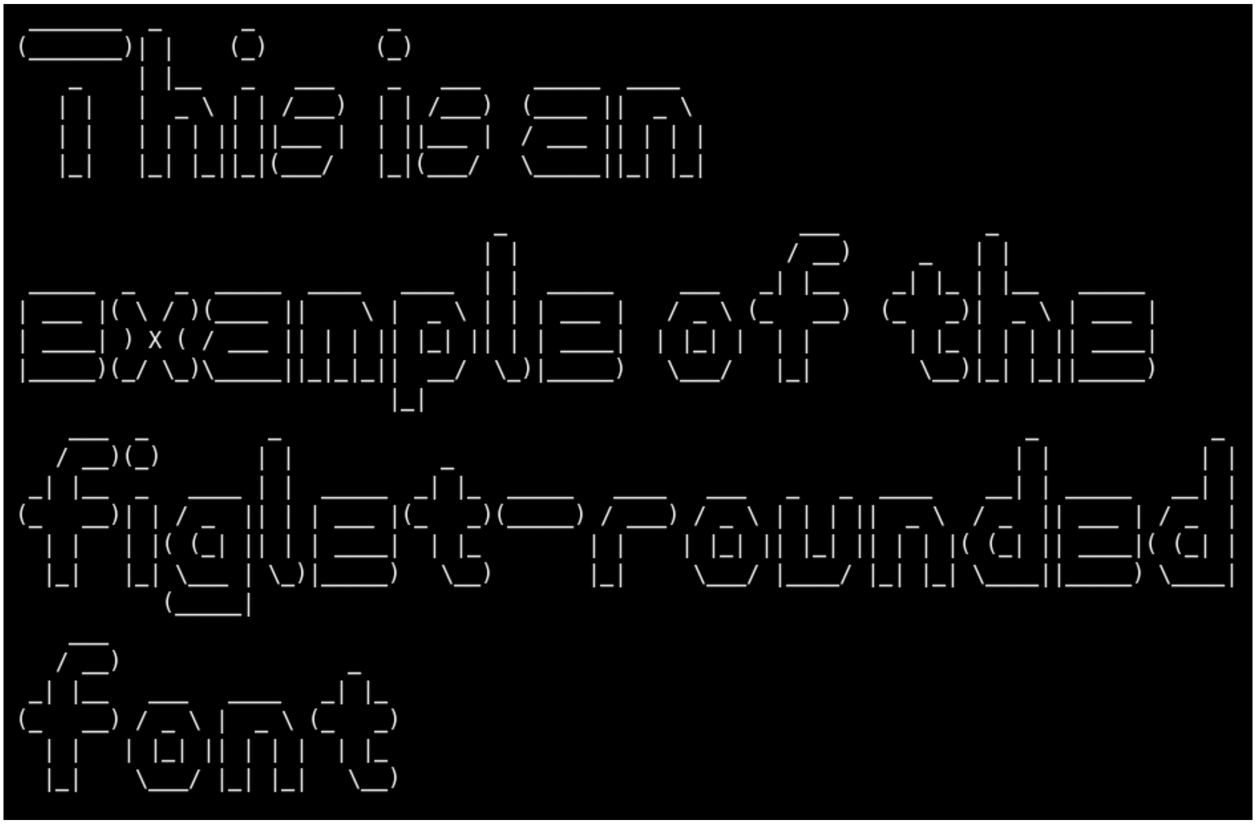
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-rounded")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by “figlet-” are but a small portion of what’s available here.

To easily convert FIGlet fonts to the pygamelib’s format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-stampatello

New in version 1.3.0.

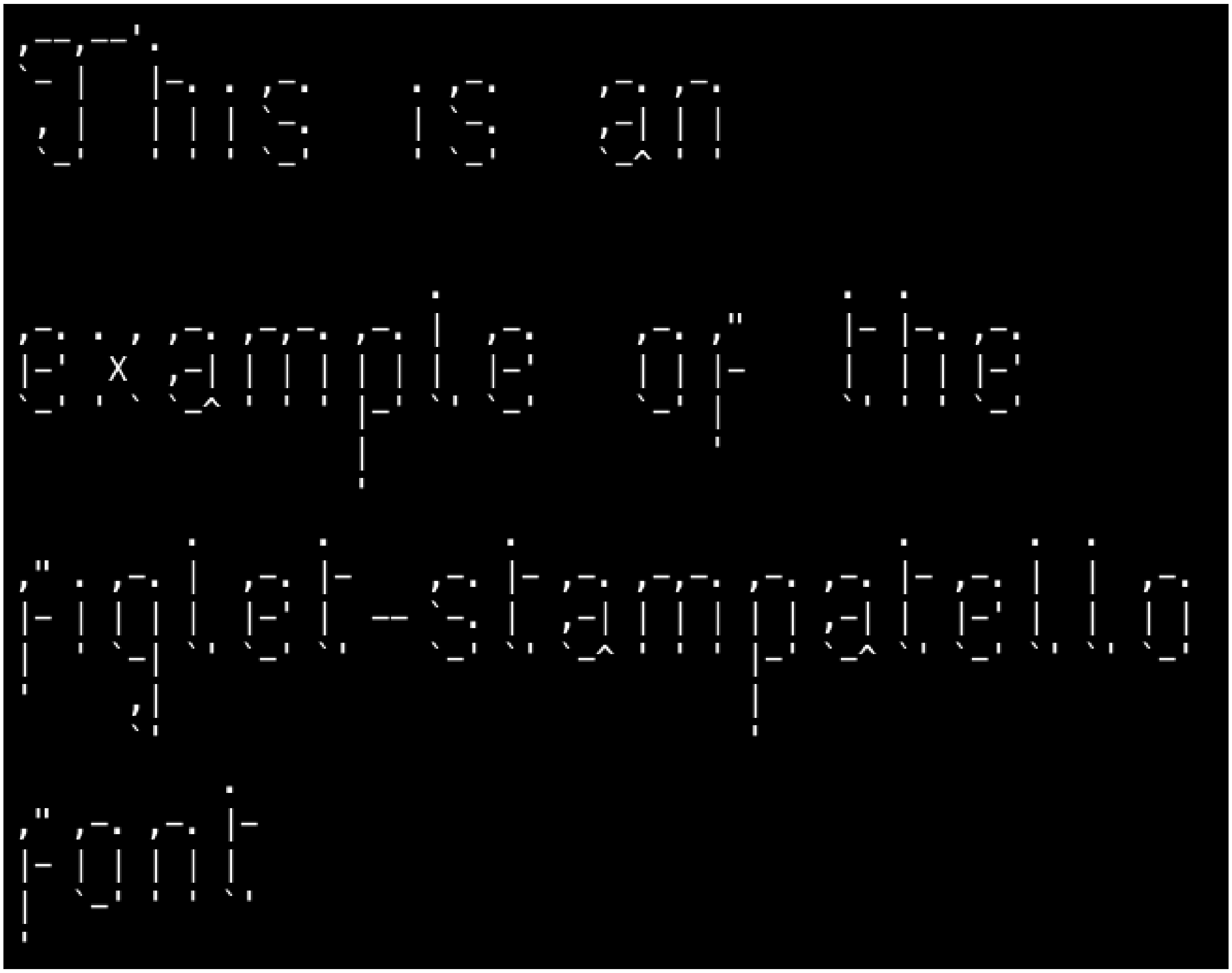
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-stampatello")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
,--,--'.
`- |   |-. . ,-.     . ,-.     ,-. ,-.
 , |   | | | `-.     | `-.     ,-| | |
 `-'   ' ' ' `-'     ' `-'     `-^ ' '


                      .                     .  .
,-. . , ,-. ,-,-. ,-. |  ,-.     ,-. ,"     |- |-. ,-.
|-'  X  ,-| | | | | | |  |-'     | | |-     |  | | |-'
`-' ' ` `-^ ' ' ' |-' `' `-'     `-' |      `' ' ' `-'
                  |                  |
                  '


         .      .         .                    .      .  .
," . ,-. |  ,-. |-    ,-. |- ,-. ,-,-. ,-. ,-. |- ,-. |  |  ,-.
|- | | | |  |-' |  -- `-. |  ,-| | | | | | ,-| |  |-' |  |  | |
|  ' `-| `' `-' `'    `-' `' `-^ ' ' ' |-' `-^ `' `-' `' `' `-'
'     ,|                               |
      `'                               '


           .
," ,-. ,-. |-
|- | | | | |
|  `-' ' ' `'
'
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-univers

New in version 1.3.0.

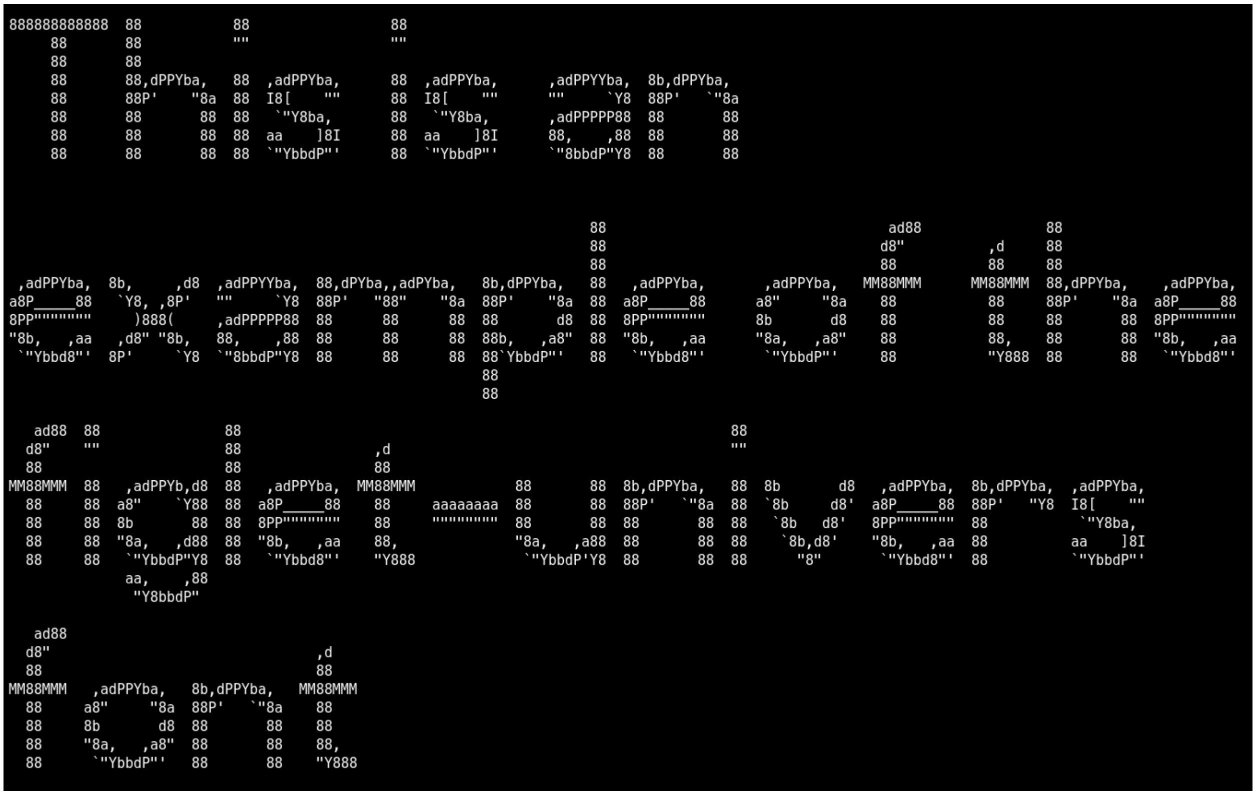
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-univers")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
888888888888  88           88                 88
     88       88           ""                 ""
     88       88
     88       88,dPPYba,   88  ,adPPYba,      88  ,adPPYba,      ,adPPYYba,  8b,dPPYba,
     88       88P'    "8a  88  I8[    ""      88  I8[    ""      ""     `Y8  88P'   `"8a
     88       88       88  88   `"Y8ba,       88   `"Y8ba,       ,adPPPPP88  88       88
     88       88       88  88  aa    ]8I      88  aa    ]8I      88,    ,88  88       88
     88       88       88  88  `"YbbdP"'      88  `"YbbdP"'      `"8bbdP"Y8  88       88



                                                                      88                                  ad88               88
                                                                      88                                 d8"          ,d     88
                                                                      88                                 88           88     88
 ,adPPYba,  8b,     ,d8  ,adPPYYba,  88,dPYba,,adPYba,   8b,dPPYba,   88   ,adPPYba,       ,adPPYba,   MM88MMM      MM88MMM  88,dPPYba,    ,adPPYba,
a8P_____88   `Y8, ,8P'   ""     `Y8  88P'   "88"    "8a  88P'    "8a  88  a8P_____88      a8"     "8a    88           88     88P'    "8a  a8P_____88
8PP"""""""     )888(     ,adPPPPP88  88      88      88  88       d8  88  8PP"""""""      8b       d8    88           88     88       88  8PP"""""""
"8b,   ,aa   ,d8" "8b,   88,    ,88  88      88      88  88b,   ,a8"  88  "8b,   ,aa      "8a,   ,a8"    88           88,    88       88  "8b,   ,aa
 `"Ybbd8"'  8P'     `Y8  `"8bbdP"Y8  88      88      88  88`YbbdP"'   88   `"Ybbd8"'       `"YbbdP"'     88           "Y888  88       88   `"Ybbd8"'
                                                         88
                                                         88

   ad88  88               88                                                           88
  d8"    ""               88                ,d                                         ""
  88                      88                88
MM88MMM  88   ,adPPYb,d8  88   ,adPPYba,  MM88MMM            88       88  8b,dPPYba,   88  8b       d8   ,adPPYba,  8b,dPPYba,  ,adPPYba,
  88     88  a8"    `Y88  88  a8P_____88    88     aaaaaaaa  88       88  88P'   `"8a  88  `8b     d8'  a8P_____88  88P'   "Y8  I8[    ""
  88     88  8b       88  88  8PP"""""""    88     """"""""  88       88  88       88  88   `8b   d8'   8PP"""""""  88           `"Y8ba,
  88     88  "8a,   ,d88  88  "8b,   ,aa    88,              "8a,   ,a88  88       88  88    `8b,d8'    "8b,   ,aa  88          aa    ]8I
  88     88   `"YbbdP"Y8  88   `"Ybbd8"'    "Y888             `"YbbdP'Y8  88       88  88      "8"       `"Ybbd8"'  88          `"YbbdP"'
              aa,    ,88
               "Y8bbdP"

   ad88
  d8"                                ,d
  88                                 88
MM88MMM   ,adPPYba,   8b,dPPYba,   MM88MMM
  88     a8"     "8a  88P'   `"8a    88
  88     8b       d8  88       88    88
  88     "8a,   ,a8"  88       88    88,
  88      `"YbbdP"'   88       88    "Y888
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

figlet-wavy

New in version 1.3.0.

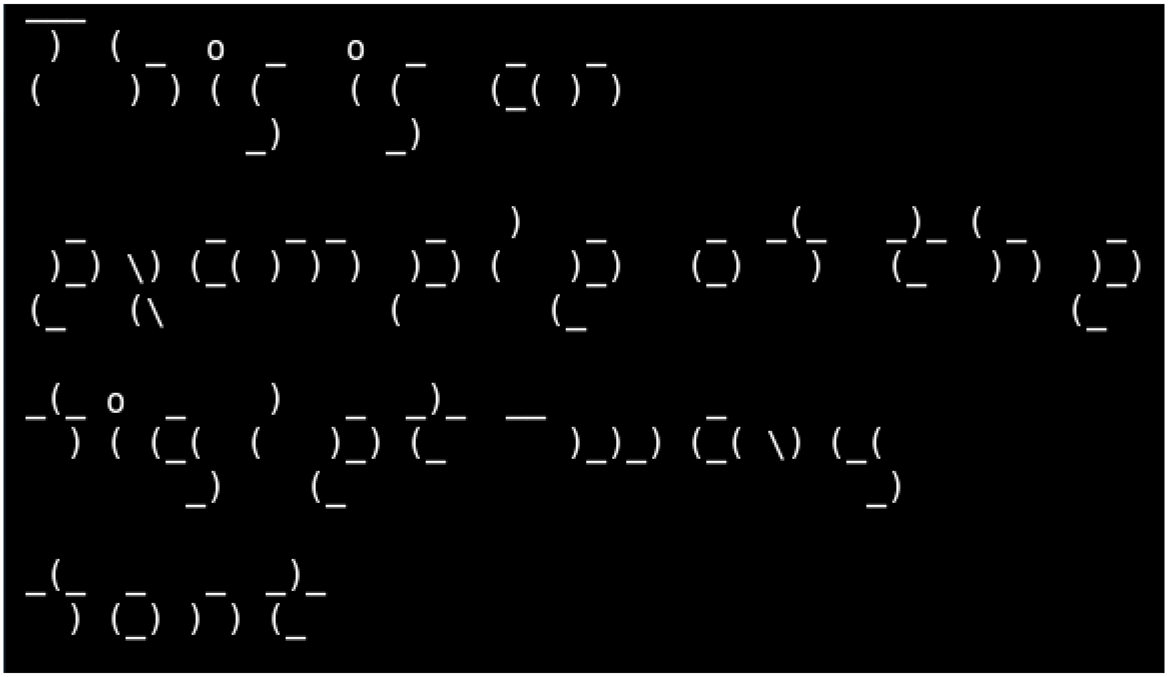
How to use?

Example:

```python
from pygamelib.gfx.core import Font
from pygamelib.base import Text

my_font = Font("figlet-wavy")
my_text = Text("Lorem Ipsum", font=my_font)
my_text.print_formatted()
```

What does it look like?



```
___
 )  ( _  o  _   o  _    _   _
(    ) ) ( (    ( (    (_( ) )
           _)     _)

  _      _   _ _    _   )   _     _  _(_   _)_ ( _    _
 )_) \) (_( ) ) )  )_) (   )_)   (_)   )   (_   ) )  )_)
(_   (\           (       (_                        (_

_(_ o  _    )   _  _)_  __        _
  ) ( (_(  (   )_) (_      )_)_) (_( \) (_(
        _)    (_                          _)

_(_  _   _  _)_
  ) (_) ) ) (_
```

More

Please visit the FIGlet project: http://www.figlet.org/

The fonts prefixed by "figlet-" are but a small portion of what's available here.

To easily convert FIGlet fonts to the pygamelib's format you can you the figlet-to-pygamelib script available here: https://github.com/pygamelib/figlet-to-pygamelib

3.3 base

The base module provide basic objects and exceptions that are used by the entire library.

3.3.1 Console

class `pygamelib.base.`**`Console`**

Bases: `object`

The Console class is a singleton wrapper around the blessed.Terminal() class. Since the library is using Terminal a lot, it is both useful and efficient to have a quick access to a single instance of the class.

This class only expose one method: *`instance()`* that returns the singleton instance.

Methods

`instance`()	Returns the instance of the blessed.Terminal object.

classmethod instance()

Returns the instance of the blessed.Terminal object.

New in version 1.3.0.

The pygamelib extensively use the Terminal object from the blessed module. However we find ourselves in need of a Terminal instance a lot, so to help with memory and execution time we just encapsulate the Terminal object in a singleton so any object can use it without instantiating it many times (and messing up with the contexts).

Returns
Instance of blessed.Terminal object

Example:

```
term = Console.instance()
```

3.3.2 History

class pygamelib.base.**History**

Bases: `object`

New in version 1.4.0.

History is a general history management object. It works by managing a time structure with past, current and future actions. Undoing and redoing is simply the action of moving along this timeline.

This object is data agnostic, it can be used with any type of objects. Internally, it uses a concept of "actions". An "action" consist of any python object that can be stored in an array.

Undoing or redoing over the limit of the past and future actions is not doing anything.

When *add()* is called, it **clears the future actions**.

This object provides a singleton interface through the *instance()*. It can be created as a normal instantiated object or as a global manager depending on your needs.

Example:

```
global_history = History.instance()
global_history.add('Hel')
global_history.add('Hello')
global_history.add('Hello Wo')
global_history.add('Hello World')
print(global_history.current)  # print "Hello World"
global_history.undo()
print(global_history.current)  # print "Hello Wo"
global_history.undo()
print(global_history.current)  # print "Hello"
global_history.redo()
global_history.redo()
global_history.redo() # This one does nothing as we called undo only twice.
print(global_history.current)  # print "Hello World"
# Now an example of reset through add()
global_history.undo()
global_history.undo()
print(global_history.current)  # print "Hello"
global_history.add("Hello there!")
print(global_history.current)  # print "Hello there!"
global_history.redo() # does nothing as the future was reset by add()
```

__init__() → None

The constructor takes no parameters. In the future, it could take some, like the maximum size of the history for example.

Methods

`__init__`()	The constructor takes no parameters.
`add`(action)	Add an action (i.e: object) to the history.
`instance`(*args, **kwargs)	Returns/creates the instance of the History object
`redo`()	Step forward into the actions' timeline.
`reset`()	Reset the history to its initial state.
`undo`()	Step backward into the actions' timeline.

Attributes

`current`	current is a read-only property to access the current action of the history.

add(*action: object*) → None

Add an action (i.e: object) to the history.

This action becomes the current one.

Adding an action reset the future actions (i.e: you cannot redo what was undone before adding the action).

Parameters

action (*object*) – The action to add to the history.

Example:

```
global_history = History.instance()
global_history.add('Hel')
global_history.add('Hello')
print(global_history.current)  # print "Hello"
```

property current: object

current is a read-only property to access the current action of the history.

To add an action and set it as current, use the `add()` method.

classmethod instance(**args*, ***kwargs*)

Returns/creates the instance of the History object

Creates an History object on first call an then returns the same instance on further calls

Returns

Instance of the History object

Return type

History

redo() → None

Step forward into the actions' timeline.

This method takes the current action and puts it in the past queue. It then pop the last action of the future queue and set it as current.

If there's no more future actions, it does nothing.

Example:

```
global_history = History.instance()
global_history.add('Hel')
global_history.add('Hello')
print(global_history.current)  # print "Hello"
global_history.undo()
print(global_history.current)  # print "Hel"
global_history.redo()
print(global_history.current)  # print "Hello"
```

reset() → None

Reset the history to its initial state. It clears all the past and future actions. It also set the current action to None.

undo() → None

Step backward into the actions' timeline.

This method takes the current action and puts it in the future queue. It then pop the last action of the past queue and set it as current.

If there's no more past actions, it does nothing.

Example:

```
global_history = History.instance()
global_history.add('Hel')
global_history.add('Hello')
print(global_history.current)  # print "Hello"
global_history.undo()
print(global_history.current)  # print "Hel"
```

3.3.3 Math

class pygamelib.base.**Math**

Bases: object

The math class regroup math functions required for game development.

New in version 1.2.0.

For the moment there is only static methods in that class but it will evolve in the future.

__init__()

Methods

__init__()	
distance(row1, column1, row2, column2)	Return the euclidean distance between to points.
intersect(row1, column1, width1, height1, ...)	This function check if 2 rectangles intersect.
lerp(a, b, t)	Return the linear interpolation between 2 values relative to a third value.

static distance(*row1: int*, *column1: int*, *row2: int*, *column2: int*) → float

Return the euclidean distance between to points.

Points are identified by their row and column. If you want the distance in number of cells, you need to round the result (see example).

Parameters

- **row1** (*int*) – the row number (coordinate) of the first point.
- **column1** (*int*) – the column number (coordinate) of the first point.
- **row2** (*int*) – the row number (coordinate) of the second point.
- **column2** (*int*) – the column number (coordinate) of the second point.

Returns
The distance between the 2 points.

Return type
float

Example:

```
distance = round(base.Math.distance(player.row,
                                player.column,
                                npc.row,
                                npc.column)
                )
```

static intersect(*row1: int*, *column1: int*, *width1: int*, *height1: int*, *row2: int*, *column2: int*, *width2: int*, *height2: int*) → bool

This function check if 2 rectangles intersect.

The 2 rectangles are defined by their positions (row, column) and dimension (width and height).

Parameters

- **row1** (*int*) – The row of the first rectangle
- **column1** (*int*) – The column of the first rectangle
- **width1** (*int*) – The width of the first rectangle
- **height1** (*int*) – The height of the first rectangle
- **row2** (*int*) – The row of the second rectangle
- **column2** – The column of the second rectangle
- **width2** (*int*) – The width of the second rectangle
- **height2** (*int*) – The height of the second rectangle

Returns
A boolean, True if the rectangles intersect False, otherwise.

Example:

```
if intersect(projectile.row, projectile.column, projectile.width,
             projectile.height, bady.row, bady.column, bady.width,
             bady.height):
    projectile.hit([bady])
```

static lerp(*a: float*, *b: float*, *t: float*) → float

Return the linear interpolation between 2 values relative to a third value.

New in version 1.3.0.

Parameters

- **a** (*float*) – Start value of the interpolation. Returned if t is 0.
- **b** (*float*) – End value of the interpolation. Returned if t is 1.
- **t** (*float*) – A value between 0 and 1 used to interpolate between a and b.

Example:

```
value = lerp(0, 100, 0.5) # 50
```

3.3.4 PglBaseObject

class pygamelib.base.**PglBaseObject**

Bases: `object`

The base object of most of the pygamelib's classes.

New in version 1.3.0.

The PglBaseObject has 2 goals:

- Store the object's screen position.
- Implements a modified observer design pattern.

It is "modified" as it acts both as the observer and the client. The idea behind it is that any object can observe and be observed by any other objects.

The base logic of the pattern is already implemented and probably does not require re-implementation on the child object. However, the *handle_notification()* method needs to be implemented in each client. The actual processing of the notification is indeed specific to each object.

Storing the screen position is particularly useful for *BoardItem* subclasses as they only know their position relative to the *Board* but might need to know their absolute screen coordinates.

This is a lightweight solution to that issue. It is not foolproof however! The screen_row and screen_column attributes are not wrapped properties and can be modified to mess up things. It shouldn't be done lightly. You have been warned!

__init__() → None

Like the object class, this class constructor takes no parameter.

Methods

__init__()	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.3.5 PglException

exception pygamelib.base.**PglException**(*error*, *message*)

Exception raised for non specific errors in the pygamelib.

3.3.6 PglInvalidLevelException

exception pygamelib.base.**PglInvalidLevelException**(*message*)

Exception raised if a level is not associated to a board in Game().

3.3.7 PglInvalidTypeException

exception pygamelib.base.**PglInvalidTypeException**(*message*)

Exception raised for invalid types.

3.3.8 PglInventoryException

exception pygamelib.base.**PglInventoryException**(*error*, *message*)

Exception raised for issue related to the inventory. The error is an explicit string, and the message explains the error.

3.3.9 PglObjectIsNotMovableException

exception pygamelib.base.**PglObjectIsNotMovableException**(*message*)

Exception raised if the object that is being moved is not a subclass of Movable.

3.3.10 PglOutOfBoardBoundException

exception pygamelib.base.**PglOutOfBoardBoundException**(*message*)

Exception for out of the board's boundaries operations.

3.3.11 Text

class pygamelib.base.**Text**(*text=''*, *fg_color=None*, *bg_color=None*, *style=''*, *font=None*)

Bases: *PglBaseObject*

An object to manipulate and display text in multiple contexts.

New in version 1.2.0.

The Text class is a collection of text formatting and display static methods.

You can either instantiate an object or use the static methods.

The Text object allow for easy text manipulation through its collection of independent attributes. They help to set the text, its style and the foreground and background colors.

The Text object can be converted to a *Sprite* through the Sprite.from_text() method. This is particularly useful to the place text on the game *Board*.

__init__(*text=''*, *fg_color=None*, *bg_color=None*, *style=''*, *font=None*)

Parameters

- **text** (*str*) – The text to manipulate
- **fg_color** (*Color*) – The foreground color for the text.
- **bg_color** (*Color*) – The background color for the text.
- **style** (*str*) – The style for the text.
- **font** (*Font*) – The font in which the text is going to be displayed (only works when using Screen.place() and Screen.update())

Methods

__init__([text, fg_color, bg_color, style, font])	**param text** The text to manipulate
attach(observer)	Attach an observer to this instance.
handle_notification(target[, attribute, value])	A virtual method that needs to be implemented by the observer.
black(message)	This method works exactly the way green_bright() work with different color.
black_bright(message)	This method works exactly the way green_bright() work with different color.
black_dim(message)	This method works exactly the way green_bright() work with different color.
blue(message)	This method works exactly the way green_bright() work with different color.
blue_bright(message)	This method works exactly the way green_bright() work with different color.
blue_dim(message)	This method works exactly the way green_bright() work with different color.
cyan(message)	This method works exactly the way green_bright() work with different color.

continues on next page

Table 6 – continued from previous page

cyan_bright(message)	This method works exactly the way green_bright() work with different color.
cyan_dim(message)	This method works exactly the way green_bright() work with different color.
debug(message)	Print a debug message.
detach(observer)	Detach an observer from this instance.
fatal(message)	Print a fatal message.
green(message)	This method works exactly the way green_bright() work with different color.
green_bright(message)	Return a string formatted to be bright green
green_dim(message)	This method works exactly the way green_bright() work with different color.
info(message)	Print an informative message.
magenta(message)	This method works exactly the way green_bright() work with different color.
magenta_bright(message)	This method works exactly the way green_bright() work with different color.
magenta_dim(message)	This method works exactly the way green_bright() work with different color.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
print_white_on_red(message)	Print a white message over a red background.
red(message)	This method works exactly the way green_bright() work with different color.
red_bright(message)	This method works exactly the way green_bright() work with different color.
red_dim(message)	This method works exactly the way green_bright() work with different color.
render_to_buffer(buffer, row, column, ...)	Render the Text object from the display buffer to the frame buffer.
warn(message)	Print a warning message.
white(message)	This method works exactly the way green_bright() work with different color.
white_bright(message)	This method works exactly the way green_bright() work with different color.
white_dim(message)	This method works exactly the way green_bright() work with different color.
yellow(message)	This method works exactly the way green_bright() work with different color.
yellow_bright(message)	This method works exactly the way green_bright() work with different color.
yellow_dim(message)	This method works exactly the way green_bright() work with different color.

Attributes

bg_color	The bg_color attribute sets the background color.
fg_color	The fg_color attribute sets the foreground color.
length	Return the true length of the text.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
text	The text attribute.
style	The style attribute sets the style of the text.
parent	This object's parent.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property bg_color

The bg_color attribute sets the background color. It needs to be a *Color*.

New in version 1.3.0.

When the background color is changed, the observers are notified of the change with the pygamelib.base.Text.bg_color:changed event. The new color is passed as the *value* parameter.

static black(*message*)

This method works exactly the way green_bright() work with different color.

static black_bright(*message*)

This method works exactly the way green_bright() work with different color.

static black_dim(*message*)

This method works exactly the way green_bright() work with different color.

static blue(*message*)

This method works exactly the way green_bright() work with different color.

static blue_bright(*message*)

This method works exactly the way green_bright() work with different color.

static blue_dim(*message*)

This method works exactly the way green_bright() work with different color.

static cyan(*message*)

This method works exactly the way green_bright() work with different color.

static cyan_bright(*message*)

This method works exactly the way green_bright() work with different color.

static cyan_dim(*message*)

This method works exactly the way green_bright() work with different color.

static debug(*message*)

Print a debug message.

The debug message is a regular message prefixed by INFO in blue on a green background.

Parameters
message (*str*) – The message to print.

Example:

```
base.Text.debug("This is probably going to success, eventually...")
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

static fatal(*message*)

Print a fatal message.

The fatal message is a regular message prefixed by FATAL in white on a red background.

Parameters
message (*str*) – The message to print.

Example:

```
base.Text.fatal("|x_x|")
```

property fg_color

The fg_color attribute sets the foreground color. It needs to be a Color.

New in version 1.3.0.

When the foreground color is changed, the observers are notified of the change with the pygamelib.base.Text.fg_color:changed event. The new color is passed as the *value* parameter.

static **green**(*message*)

This method works exactly the way green_bright() work with different color.

static **green_bright**(*message*)

Return a string formatted to be bright green

Parameters
`message` (*str*) – The message to format.

Returns
The formatted string

Return type
str

Example:

```
print( Text.green_bright("This is a formatted message") )
```

static **green_dim**(*message*)

This method works exactly the way green_bright() work with different color.

handle_notification(*target*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

static **info**(*message*)

Print an informative message.

The info is a regular message prefixed by INFO in white on a blue background.

Parameters
`message` (*str*) – The message to print.

Example:

```
base.Text.info("This is a very informative message.")
```

property **length**

Return the true length of the text.

New in version 1.3.0.

With UTF8 and emojis the length of a string as returned by python's `len()` function is often very wrong. For example, the len("x1b[48;2;139;22;19mx1b[38;2;160;26;23mx1b[0m") returns 39 when it should return 1.

This method returns the actual printing/display size of the text.

Note: This is a read only value. It is automatically updated when the text property is changed.

Example:

```
game.screen.place(my_text, 0, game.screen.width - my_text.length)
```

classmethod load(*data: dict = None*)

Load data and create a new Text object out of it.

New in version 1.3.0.

Parameters

data (*dict*) – Data to create a new actuator (usually generated by *serialize()*)

Returns

A new Text object.

Return type

Text

Example:

```
title = base.Text.load( previous_title.serialize() )
```

static magenta(*message*)

This method works exactly the way green_bright() work with different color.

static magenta_bright(*message*)

This method works exactly the way green_bright() work with different color.

static magenta_dim(*message*)

This method works exactly the way green_bright() work with different color.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

parent

This object's parent. It needs to be a *BoardItem*.

print_formatted()

Print the text with the current font activated.

New in version 1.3.0.

If the font is not set, it is strictly equivalent to use Python's print(text_object).

static **print_white_on_red**(*message*)

Print a white message over a red background.

Parameters
message (*str*) – The message to print.

Example:

```
base.Text.print_white_on_red("This is bright!")
```

static **red**(*message*)

This method works exactly the way green_bright() work with different color.

static **red_bright**(*message*)

This method works exactly the way green_bright() work with different color.

static **red_dim**(*message*)

This method works exactly the way green_bright() work with different color.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render the Text object from the display buffer to the frame buffer.

New in version 1.3.0.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Return a dictionary with all the attributes of this object.

New in version 1.3.0.

Returns
A dictionary with all the attributes of this object.

Return type
dict

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

style

The style attribute sets the style of the text. It needs to be a str.

property **text**

The text attribute. It needs to be a str.

New in version 1.3.0.

When the text is changed, the observers are notified of the change with the pygamelib.base.Text.text:changed event. The new text is passed as the *value* parameter.

static **warn**(*message*)

Print a warning message.

The warning is a regular message prefixed by WARNING in black on a yellow background.

Parameters
message (*str*) – The message to print.

Example:

```
base.Text.warn("This is a warning.")
```

static **white**(*message*)

This method works exactly the way green_bright() work with different color.

static **white_bright**(*message*)

This method works exactly the way green_bright() work with different color.

static **white_dim**(*message*)

This method works exactly the way green_bright() work with different color.

static **yellow**(*message*)

This method works exactly the way green_bright() work with different color.

static yellow_bright(*message*)

This method works exactly the way green_bright() work with different color.

static yellow_dim(*message*)

This method works exactly the way green_bright() work with different color.

3.3.12 Vector2D

class pygamelib.base.**Vector2D**(*row=0.0*, *column=0.0*)

Bases: `object`

A 2D vector class.

New in version 1.2.0.

Contrary to the rest of the library Vector2D uses floating point numbers for its coordinates/direction/orientation. However since the rest of the library uses integers, the numbers are rounded to 2 decimals. You can alter that behavior by increasing or decreasing the rounding_precision parameter (if you want integer for example).

Vector2D use the row/column internal naming convention as it is easier to visualize for developers that are still learning python or the pygamelib. If it is a concept that you already understand and are more familiar with the x/y coordinate system you can also use x and y.

- x is equivalent to column
- y is equivalent to row

Everything else is the same.

Vectors can be printed and supports basic operations:

- addition
- substraction
- multiplication

Let's elaborate a bit more on the multiplication. The product behaves in 2 different ways:

If you multiply a vector with a scalar (int or float), the return value is a Vector2D with each vector component multiplied by said scalar.

If you multiply a Vector2D with another Vector2D you ask for the the cross product of vectors. This is an undefined mathematical operation in 2D as the cross product is supposed to be perpendicular to the 2 other vectors (along the z axis in our case). Since we don't have depth (z) in 2D, this will return the magnitude of the signed cross product of the 2 vectors.

Example of products:

```
v1 = base.Vector2D(1,2)
v2 = base.Vector2D(3,4)
# This returns -2
mag = v1 * v2
# This returns a Vector2D with values (-1, -2)
inv = v1 * -1
# This return a Vector2D with values (2.85, 3.8) or 95% of v2
dim = v2 * 0.95
```

Parameters

- **row** (*int*) – The row/y parameter.
- **column** (*int*) – The column/x parameter.

Example:

```
gravity = Vector2D(9.81, 0)
# Remember that minus on row is up.
speed = Vector2D(-0.123, 0.456)
# In that case you might want to increase the rounding precision
speed.rounding_precision = 3
```

__init__(*row=0.0*, *column=0.0*)

Methods

__init__([row, column])	
from_direction(direction, step)	Build and return a Vector2D from a direction.
length()	Returns the length of a vector.
load(data)	Loads a vector from a dictionary.
serialize()	Returns a dictionary with the attributes of the vector.
unit()	Returns a normalized unit vector.

Attributes

column	The column component of the vector.
row	The row component of the vector.
x	x is an alias for column.
y	y is an alias for row.
rounding_precision	The rounding_precision attributes is used when vectors values are calculated and the result rounded for convenience.

property column

The column component of the vector.

classmethod from_direction(*direction: Direction*, *step*)

Build and return a Vector2D from a direction.

Directions are from the constants module.

Parameters

- **direction** (*Direction*) – A direction from the constants module.
- **step** (*int*) – The number of cell to cross in one movement.

Example:

```
v2d_up = Vector2D.from_direction(Direction.UP, 1)
```

length()

Returns the length of a vector.

Return type
float

Example:

```
if speed.length() == 0.0:
    print('We are not moving... at all...')
```

classmethod load(*data*)

Loads a vector from a dictionary.

New in version 1.3.0.

Parameters
data (*dict*) – A dictionary with the attributes of the vector.

Returns
A vector.

Return type
Vector2D

Example:

```
gravity_dict = {"row": 9.81, "column": 0}
gravity = Vector2D.load(gravity_dict)
```

rounding_precision

The rounding_precision attributes is used when vectors values are calculated and the result rounded for convenience. It can be changed anytime to increase or decrease the precision anytime.

property row

The row component of the vector.

serialize()

Returns a dictionary with the attributes of the vector.

New in version 1.3.0.

Returns
A dictionary with the attributes of the vector.

Return type
dict

Example:

```
gravity = Vector2D(9.81, 0)
gravity_dict = gravity.serialize()
print(gravity_dict)
```

unit()

Returns a normalized unit vector.

Returns
A unit vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

property x

x is an alias for column.

property y

y is an alias for row.

3.3.13 Deprecated objects

These are the deprecated objects of the pygamelib.base module. They should not be used as they are going to be removed in future versions.

HacException

exception pygamelib.base.**HacException**(*error*, *message*)

A simple forward to PglException

Deprecated since version 1.3.0.

HacInvalidLevelException

exception pygamelib.base.**HacInvalidLevelException**(*message*)

Forward to PglInvalidLevelException

Deprecated since version 1.3.0.

HacInvalidTypeException

exception pygamelib.base.**HacInvalidTypeException**(*message*)

A simple forward to PglInvalidTypeException

Deprecated since version 1.3.0.

HacObjectIsNotMovableException

exception pygamelib.base.**HacObjectIsNotMovableException**(*message*)

Simple forward to PglObjectIsNotMovableException

Deprecated since version 1.3.0.

HacOutOfBoardBoundException

exception pygamelib.base.**HacOutOfBoardBoundException**(*message*)

Simple forward to PglOutOfBoardBoundException

Deprecated since version 1.3.0.

3.4 board_items

3.4.1 Actionable

class pygamelib.board_items.**Actionable**(*action=None*, *action_parameters=None*, *perm=None*, ***kwargs*)

Bases: `Immovable`

This class derives `Immovable`. It adds the ability to an Immovable BoardItem to be triggered and execute some code.

If an actionable board item is activated by an item (this mechanism is taken care of by the Board class), the function passed as the *action* parameter is called with *action_parameters* as parameters. Subclass may implement a different mechanism for activation so please read their documentations.

Parameters

- **action** (*function*) – the reference to a function (Attention: no parentheses at the end of the function name). It needs to be callable.
- **action_parameters** (*list*) – the parameters to the action function.
- **perm** (`constants`) – The permission that defines what types of items can actually activate the actionable. The permission has to be one of the permissions defined in `constants`. By default it is set to Permission.PLAYER_AUTHORIZED.

On top of these parameters Actionable accepts all parameters from `Immovable` and therefor from `BoardItem`.

Note: The common way to use this class is to use GenericActionableStructure. Please refer to `GenericActionableStructure` for more details.

Important: There's a complete tutorial about Actionable items on the pygamelib wiki

__init__(*action=None*, *action_parameters=None*, *perm=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([action, action_parameters, perm])	Like the object class, this class constructor takes no parameter.
`activate`()	This function is calling the action function with the action_parameters.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load data and create a new BoardItem out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

activate()

This function is calling the action function with the action_parameters.

The *action* callback function should therefor have a signature like:

```
def my_callback_function(actionable, action_parameters)
```

With *actionable* being the Actionable current reference to *self*.

Usually it's automatically called by *move()* when a Player or NPC (see `board_items`)

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset:* Vector2D = *None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns

A dictionary with all the attributes of this object.

Return type

dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.2 ActionableTile

class pygamelib.board_items.**ActionableTile**(***kwargs*)

Bases: *Actionable*, *Tile*

The ActionableTile is the complex (i.e: multi-cells items) version of the *GenericActionableStructure*. It allows you to create any type of in game object that is represented with more than one character in the terminal and that is *Actionable*. Actionable object have a callback system that is automatically called when the player collide with the object.

Important: There's a complete tutorial about Actionable items on the pygamelib wiki

__init__(***kwargs*)

Please have a look at the documentation for *Tile* and *Actionable* for the list of possible constructor's parameters.

Methods

`__init__`(**kwargs)	Please have a look at the documentation for `Tile` and `Actionable` for the list of possible constructor's parameters.
`activate`()	This function is calling the action function with the action_parameters.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	A Tile cannot move.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new Tile out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

activate()

This function is calling the action function with the action_parameters.

The *action* callback function should therefor have a signature like:

```
def my_callback_function(actionable, action_parameters)
```

With *actionable* being the Actionable current reference to *self*.

Usually it's automatically called by *move()* when a Player or NPC (see `board_items`)

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

A Tile cannot move.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new Tile out of it.

Parameters
data (*dict*) – Data to create a new tile (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.Tile

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns

A dictionary with all the attributes of this object.

Return type

dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.3 BoardComplexItem

class pygamelib.board_items.**BoardComplexItem**(*sprite=None*, *size=None*, *null_sprixel=None*, *base_item_type=None*, ***kwargs*)

Bases: *BoardItem*

New in version 1.2.0.

A BoardComplexItem is the base item for multi cells elements. It inherits from *BoardItem* and accepts all its parameters.

The main difference is that a complex item can use *Sprite* as representation.

You can see a complex item as a collection of other items that are ruled by the same laws. They behave as one but a complex item is actually made of complex components. At first it is not important but you may want to exploit that as a feature for your game.

On top of *BoardItem* the constructor accepts the following parameters:

Parameters

- **sprite** (*Sprite*) – A sprite representing the item.
- **size** (*array[int]*) – The size of the item as [WIDTH, HEIGHT]. It impact movement and collision detection amongst other things. If it is left empty the Sprite size is used. If no sprite is given to the constructor the default size is 2x2.
- **base_item_type** (*BoardItemComplexComponent*) – the building block of the complex item. The complex item is built from a 2D array of base items.

Null_sprixel

The null_sprixel is a bit of a special parameter: during construction a null sprixel is replaced by a BoardItemVoid. This is a trick to show the background (i.e transparency). A sprixel can take the color of the background but a complex item with a null_sprixel that correspond to transparent zone of a sprite will really be transparent and show the background.

Null_sprixel

Sprixel

__init__(*sprite=None*, *size=None*, *null_sprixel=None*, *base_item_type=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__([sprite, size, null_sprixel, ...])	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Returns True if the item can move, False otherwise.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
item(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
load(data)	Load data and create a new BoardComplexItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	Returns True if the item is pickable, False otherwise.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.
update_sprite()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Returns True if the item can move, False otherwise.

Example:

```
if board.item(4,5).can_move():
    print('The item can move')
```

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type

bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns

The column coordinate

Return type

int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type

str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardComplexItem out of it.

Parameters
data (*dict*) – Data to create a new complex item (usually generated by *serialize()*)

Returns
A new complex item.

Return type
~pygamelib.board_items.BoardComplexItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
`value` (*int*) – the screen column

Return type
int

`property screen_row: int`

A property to get/set the screen row.

Parameters
`value` (*int*) – the screen row

Return type
int

`serialize()` → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

`set_can_move`(*value*)

Set the value of the can_move property to value.

Parameters
`value` (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

`set_overlappable`(*value*)

Set the value of the overlappable property to value.

Parameters
`value` (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

`set_pickable`(*value*)

Set the value of the pickable property to value.

Parameters
`value` (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

`set_restorable`(*value*)

Set the value of the restorable property to value.

Parameters
`value` (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.

- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.4 BoardItemComplexComponent

class pygamelib.board_items.**BoardItemComplexComponent**(***kwargs*)

Bases: *BoardItem*

The default component of a complex item.

It is literally just a BoardItem but is subclassed for easier identification.

It is however scanning its parent for the item’s basic properties (overlappable, restorable, etc.)

A component can never be pickable by itself.

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__(**kwargs)	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Returns True if the item can move, False otherwise.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new BoardItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	Returns False.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Returns True if the item can move, False otherwise.

Example:

```
if board.item(4,5).can_move():
    print('The item can move')
```

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns

A new item.

Return type

~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns False. A component is never pickable by itself (either the whole complex item is pickable or not, but not partially)

Example:

```
if item.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns

The position as a 2D vector

Return type

Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```python
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```python
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.5 BoardItem

class pygamelib.board_items.**BoardItem**(*sprixel=None*, *model=None*, *name=None*, *item_type=None*, *parent=None*, *pickable=False*, *overlappable=False*, *restorable=False*, *can_move=False*, *pos=None*, *value=None*, *inventory_space=1*, *animation: Animation = None*, *particle_emitter=None*)

Bases: `PglBaseObject`

Base class for any item that will be placed on a Board.

Parameters

- **type** (*str*) – A type you want to give your item. It can be any string. You can then use the type for sorting or grouping for example.
- **name** (*str*) – A name for this item. For identification purpose.
- **pos** (*list*) – the position of this item. When the item is managed by the Board and Game engine this member hold the last updated position of the item. It is not updated if you manually move the item. It must be an array of 2 integers [row,column]
- **model** (*str*) – The model to use to display this item on the Board. Be mindful of the space it will require. Default value is '*'. This parameter is now deprecated in favor of "sprixel". If both "sprixel" and "model" are specified, "model" is ignored.
- **parent** – The parent object of the board item. Usually a Board or Game object.
- **sprixel** (*Sprixel*) – The sprixel that will represent the item on the Board.
- **pickable** (*bool*) – Represent the capacity for a BoardItem to be pick-up by player or NPC. This parameter is True or False. If sets to None, it'll be set to False.
- **overlappable** (*bool*) – Represent to be overlapped by another BoardItem. This parameter is True or False. If sets to None, it'll be set to False.
- **restorable** (*bool*) – Represent the capacity for an Immovable BoardItem to be restored by the board if the item is overlappable and has been overlapped by another BoardItem. This parameter is True or False. If sets to None, it'll be set to False.
- **can_move** (*bool*) – Represent the ability of the BoardItem to move on the Board. If this parameter is False, the Board.move() method will not allow the item to move. This parameter is True or False. If sets to None, it'll be set to False.
- **pos** – The position of the BoardItem on a `Board`. Please make sure that you understand what you do before changing that parameter. The position of an item is managed by the Board object and will be updated. In most cases you don't need to use that parameter. The position is a list of 2 or 3 int: [row, column, layer].
- **value** (*int | float*) – The value of an item. It can be used for any game purpose: a score indicator, a trade value, the amount of XP to grant to a player on a kill, etc.
- **inventory_space** (*int*) – The space that the item takes in the `pygamelib.engine.Inventory`. This parameter used to be available only for `Immovable` items but since 1.3.0, every BoardItem can be configured to be pickable, so every BoardItem can now take space in the inventory. Default value is 1.
- **animation** (*Animation*) – An animation to animate the item sprixel.
- **particle_emitter** (*ParticleEmitter*) – A particle emitter that is attached to this item.

Note: Starting with version 1.2.0 and introduction of complex items, BoardItems have a size. That size **CANNOT** be set. It is always 1x1. This is because a BoardItem always takes 1 cell, regardless of its actual number of characters. The size is a read-only property.

Important: In version 1.3.0 the BoardItem object has been reworked to make sure that the pickable, restorable, overlappable and can_move properties are configurable for all items independently of their type. This fixes an issue with restorable: only *Immovable* objects could be restorable. Now all items can be any combination of these properties. As a developer you are now encouraged to use the corresponding functions to determine the abilities of an item.

Warning: An item cannot be restorable and pickable at the same time. If it's pickable, it's put into the inventory of the item overlapping it. Therefor, it cannot be restored. If both restorable and pickable are set to True, one of the 2 is set to False depending on the value of overlappable: if True restorable is set to True and pickable to False and the contrary if overlappable is False.

`__init__`(*sprixel=None*, *model=None*, *name=None*, *item_type=None*, *parent=None*, *pickable=False*, *overlappable=False*, *restorable=False*, *can_move=False*, *pos=None*, *value=None*, *inventory_space=1*, *animation:* Animation = *None*, *particle_emitter=None*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([sprixel, model, name, item_type, ...])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Returns True if the item can move, False otherwise.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load data and create a new BoardItem out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Returns True if the item can move, False otherwise.

Example:

```python
if board.item(4,5).can_move():
    print('The item can move')
```

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type

bool

Example:

```python
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns

The column coordinate

Return type

int

Example:

```python
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type

str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

`display()`

Print the model WITHOUT carriage return.

`distance_to(`*other*`)`

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

`handle_notification(`*subject*, *attribute=None*, *value=None*`)`

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

`property heading`

Return the heading of the item.

This is a read only property that is updated by `store_position()`.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

`set_can_move`(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

`set_overlappable`(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

`set_pickable`(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

`set_restorable`(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

`property size`

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.6 BoardItemVoid

class pygamelib.board_items.**BoardItemVoid**(***kwargs*)

Bases: *BoardItem*

A class that represent a void cell.

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__(**kwargs)	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Returns True if the item can move, False otherwise.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new BoardItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	A BoardItemVoid is obviously overlappable (so player and NPC can walk over).
pickable()	A BoardItemVoid is not pickable, therefor this method return false.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Returns True if the item can move, False otherwise.

Example:

```
if board.item(4,5).can_move():
    print('The item can move')
```

collides_with(*other*, *projection_offset:* Vector2D = *None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type

bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns

The column coordinate

Return type

int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type

str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

A BoardItemVoid is obviously overlappable (so player and NPC can walk over).

Returns
True

property particle_emitter

pickable()

A BoardItemVoid is not pickable, therefor this method return false.

Returns
False

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.7 Camera

class pygamelib.board_items.**Camera**(*actuator=None*, ***kwargs*)

Bases: *Movable*

New in version 1.3.0.

A Camera is a special item: it does not appear on the Board and actually is not even registered on it. It is only an item that you can center the board on (when using partial display). It helps for cut scenes for example.

The main difference with a regular BoardItem is that the row and column properties are writable. This means that you can directly manipulate its coordinates and partially render a huge board around that focal point.

The *Screen* buffer rendering system introduced in version 1.3.0 require a board item to be declared as the focus point of the board if partial display is enabled.

The Camera object inherits from Movable and can accept an actuator parameter. However, it is up to the developer to activate the actuators mechanics as the Camera object does not register as a NPC or a Player. The support for actuators is mainly thought for pre-scripted cut-scenes.

Example:

```
# This example leverage the Screen buffer system introduced in v1.3.0.
# It pans the camera over a huge map. The Screen.update() method automatically
# uses the Board.partial_display_focus coordinates to adjust the displayed area.
camera = Camera()
huge_board.partial_display_focus = camera
while camera.column < huge_board.width:
    camera.column += 1
    game.screen.update()
```

__init__(*actuator=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([actuator])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Movable implements can_move().
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`has_inventory`()	This is a virtual method that must be implemented in deriving class.
`load`(data)	Load data and create a new Movable out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Serialize the Immovable object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory() → bool

This is a virtual method that must be implemented in deriving class. This method has to return True or False. This represent the capacity for a Movable to have an inventory.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new Movable out of it.

Parameters
data (*dict*) – Data to create a new movable item (usually generated by *serialize()*)

Returns
A new complex item.

Return type
~pygamelib.board_items.Movable

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the Immovable object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.8 Character

class pygamelib.board_items.**Character**(*max_hp=None*, *hp=None*, *max_mp=None*, *mp=None*, *remaining_lives=None*, *attack_power=None*, *defense_power=None*, *strength=None*, *intelligence=None*, *agility=None*, ***kwargs*)

Bases: `Movable`

A base class for a character (playable or not)

Parameters

- **agility** (*int*) – Represent the agility of the character
- **attack_power** (*int*) – Represent the attack power of the character.
- **defense_power** (*int*) – Represent the defense_power of the character
- **hp** (*int*) – Represent the hp (Health Point) of the character
- **intelligence** (*int*) – Represent the intelligence of the character
- **max_hp** (*int*) – Represent the max_hp of the character
- **max_mp** (*int*) – Represent the max_mp of the character
- **mp** (*int*) – Represent the mp (Mana/Magic Point) of the character
- **remaining_lives** (*int*) – Represent the remaining_lives of the character. For a NPC it is generally a good idea to set that to 1. Unless the NPC is a multi phased boss.
- **strength** (*int*) – Represent the strength of the character

These characteristics are here to be used by the game logic but very few of them are actually used by the Game (*pygamelib.engine*) engine.

__init__(*max_hp=None*, *hp=None*, *max_mp=None*, *mp=None*, *remaining_lives=None*, *attack_power=None*, *defense_power=None*, *strength=None*, *intelligence=None*, *agility=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([max_hp, hp, max_mp, mp, ...])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Movable implements can_move().
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`has_inventory`()	This is a virtual method that must be implemented in deriving class.
`load`(data)	Load data and create a new Character out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Serialize the Character object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory() → bool

This is a virtual method that must be implemented in deriving class. This method has to return True or False. This represent the capacity for a Movable to have an inventory.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new Character out of it.

Parameters

data (*dict*) – Data to create a new character item (usually generated by *serialize()*)

Returns

A new character item.

Return type

~pygamelib.board_items.Character

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns

The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the Character object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.9 ComplexDoor

class pygamelib.board_items.**ComplexDoor**(***kwargs*)

Bases: *Door*, *BoardComplexItem*

New in version 1.2.0.

A complex door is nothing more than a *Door* mashed with a *BoardComplexItem*.

It supports all parameters of both with inheritance going first to Door and second to BoardComplexItem.

The main interest is of course the multiple cell representation and the Sprites support.

Example:

```
castle_door = ComplexDoor(
        sprite=sprite_castle_door
    )
```

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`(**kwargs)	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new ComplexDoor out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters

other (*BoardItem*) – The item you want to calculate the distance to.

Returns

The distance between this item and the other.

Return type

float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new ComplexDoor out of it.

Parameters
data (*dict*) – Data to create a new complex door (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.ComplexDoor

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.

- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.10 ComplexNPC

class pygamelib.board_items.**ComplexNPC**(***kwargs*)

Bases: *NPC*, *BoardComplexItem*

New in version 1.2.0.

A complex NPC is nothing more than a *NPC* mashed with a *BoardComplexItem*.

It supports all parameters of both with inheritance going first to NPC and second to BoardComplexItem.

The main interest is of course the multiple cell representation and the Sprites support.

Example:

```
player = ComplexNPC(
        name='Idiot McComplexStupid',
        sprite=npc_sprite_collection['troll_licking_stones']
    )
```

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`(**kwargs)	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Movable implements can_move().
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`has_inventory`()	Define if the NPC has an inventory.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new ComplexNPC out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Define if the NPC is pickable.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Serialize the NPC object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory()

Define if the NPC has an inventory.

This method returns false because the game engine doesn't manage NPC inventory yet but it could be in the future. It's a good habit to check the value returned by this function.

Returns
False

Return type
Boolean

Example:

```
if mynpc.has_inventory():
    print("Cool: we can pickpocket that NPC!")
else:
    print("No pickpocketing XP for us today :(")
```

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new ComplexNPC out of it.

Parameters
data (*dict*) – Data to create a new complex npc (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.ComplexNPC

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Define if the NPC is pickable.

Obviously this method always return False.

Returns
False

Return type
Boolean

Example:

```
if mynpc.pickable():
    Utils.warn("Something is fishy, that NPC is pickable"
        "but is not a Pokemon...")
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.

- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the NPC object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.11 ComplexPlayer

class pygamelib.board_items.**ComplexPlayer**(***kwargs*)

Bases: *Player*, *BoardComplexItem*

New in version 1.2.0.

A complex player is nothing more than a *Player* mashed with a *BoardComplexItem*.

It supports all parameters of both with inheritance going first to Player and second to BoardComplexItem.

The main interest is of course the multiple cell representation and the Sprites support.

Example:

```
player = ComplexPlayer(
        name='Mighty Wizard',
        sprite=sprite_collection['wizard_idle']
    )
```

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`(**kwargs)	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Movable implements can_move().
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`has_inventory`()	This method returns True (a player has an inventory).
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new ComplexPlayer out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	This method returns False (a player is obviously not pickable).
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Serialize the Character object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory()

This method returns True (a player has an inventory).

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new ComplexPlayer out of it.

Parameters
data (*dict*) – Data to create a new complex player (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.ComplexPlayer

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

This method returns False (a player is obviously not pickable).

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the Character object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.12 ComplexTreasure

class pygamelib.board_items.**ComplexTreasure**(***kwargs*)

Bases: *Treasure*, *BoardComplexItem*

New in version 1.2.0.

A complex treasure is nothing more than a *Treasure* mashed with a *BoardComplexItem*.

It supports all parameters of both with inheritance going first to Treasure and second to BoardComplexItem.

The main interest is of course the multiple cell representation and the Sprites support.

Example:

```
chest = ComplexTreasure(
        sprite=sprite_chest
    )
```

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`(**kwargs)	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new ComplexTreasure out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	This represent the capacity for a Treasure to be overlapped by player or NPC.
`pickable`()	This represent the capacity for a Treasure to be picked-up by player or NPC.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	This represent the capacity for a Treasure to be restored after being overlapped.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

`property height`

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

`property inventory_space`

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

`item`(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

`property layer`

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new ComplexTreasure out of it.

Parameters
data (*dict*) – Data to create a new complex treasure (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.ComplexTreasure

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

This represent the capacity for a Treasure to be overlapped by player or NPC.

A treasure is not overlappable.

Returns
False

Return type
bool

property particle_emitter

pickable()

This represent the capacity for a Treasure to be picked-up by player or NPC.

A treasure is obviously pickable by the player and potentially NPCs. *Board* puts the Treasure in the *Inventory* if the picker implements has_inventory()

Returns
True

Return type
bool

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

This represent the capacity for a Treasure to be restored after being overlapped.

A treasure is not overlappable, therefor is not restorable.

Returns
False

Return type
bool

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns

The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.13 ComplexWall

class pygamelib.board_items.**ComplexWall**(***kwargs*)

Bases: *Wall*, *BoardComplexItem*

New in version 1.2.0.

A complex wall is nothing more than a *Wall* mashed with a *BoardComplexItem*.

It supports all parameters of both with inheritance going first to Wall and second to BoardComplexItem.

The main interest is of course the multiple cell representation and the Sprites support.

Example:

```
wall = ComplexWall(
        sprite=sprite_brick_wall
    )
```

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`(**kwargs)	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new ComplexWall out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	This represent the capacity for a `BoardItem` to be overlapped by player or NPC.
`pickable`()	This represent the capacity for a `BoardItem` to be pick-up by player or NPC.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	This represent the capacity for an `Immovable` `Movable` item.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters

other (*BoardItem*) – The item you want to calculate the distance to.

Returns

The distance between this item and the other.

Return type

float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new ComplexWall out of it.

Parameters
data (*dict*) – Data to create a new complex wall item (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.ComplexWall

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

This represent the capacity for a *BoardItem* to be overlapped by player or NPC.

Returns
False

Return type
bool

property particle_emitter

pickable()

This represent the capacity for a *BoardItem* to be pick-up by player or NPC.

Returns
False

Return type
bool

Example:

```
if mywall.pickable():
    print('Whoaa this wall is really light... and small...')
else:
    print('Really? Trying to pick-up a wall?')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

This represent the capacity for an *Immovable* `Movable` item. A wall is not overlappable.

Returns
False

Return type
bool

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns

The size.

Return type

list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters

new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.14 Door

class pygamelib.board_items.**Door**(**kwargs*)

Bases: *GenericStructure*

A Door is a *GenericStructure* that is not pickable, overlappable and restorable. It has a value of 0 and a size of 1 by default. It is an helper class that allows to focus on game design and mechanics instead of small building blocks.

Parameters

- **model** (*str*) – The model that will represent the door on the map
- **value** (*int*) – The value of the door, it is useless in that case. The default value is 0.
- **inventory_space** (*int*) – The size of the door in the inventory. Unless you make the door pickable (I have no idea why you would do that…), this parameter is not used.
- **type** (*str*) – The type of the door. It is often used as a type identifier for your game main loop. For example: unlocked_door or locked_door.
- **pickable** (*Boolean*) – Is this door pickable by the player? Default value is False.
- **overlappable** (*Boolean*) – Is this door overlappable by the player? Default value is True.
- **restorable** (*Boolean*) – Is this door restorable after being overlapped? Default value is True.

Note: All the options from *GenericStructure* are also available to this constructor.

Example:

```
door1 = Door(model=graphics.Models.DOOR,type='locked_door')
```

__init__(**kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__(**kwargs)	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Return the capability of moving of an item.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new BoardItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	Returns True if the item is pickable, False otherwise.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset:* Vector2D = *None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

`set_can_move`(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

`set_overlappable`(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

`set_pickable`(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

`set_restorable`(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

`property size`

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.15 GenericActionableStructure

`class pygamelib.board_items.GenericActionableStructure(**kwargs)`

Bases: `GenericStructure`, `Actionable`

A GenericActionableStructure is the combination of a `GenericStructure` and an `Actionable`. It is only a helper combination.

Please see the documentation for `GenericStructure` and `Actionable` for more information.

Important: There's a complete tutorial about Actionable items on the pygamelib wiki

`__init__(**kwargs)`

Like the object class, this class constructor takes no parameter.

Methods

`__init__`(**kwargs)	Like the object class, this class constructor takes no parameter.
`activate`()	This function is calling the action function with the action_parameters.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load data and create a new BoardItem out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

activate()

This function is calling the action function with the action_parameters.

The *action* callback function should therefor have a signature like:

```
def my_callback_function(actionable, action_parameters)
```

With *actionable* being the Actionable current reference to *self*.

Usually it's automatically called by *move()* when a Player or NPC (see `board_items`)

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset:* Vector2D = *None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns

The position as a 2D vector

Return type

Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.16 GenericStructureComplexComponent

class pygamelib.board_items.**GenericStructureComplexComponent**(***kwargs*)

Bases: *GenericStructure*, *BoardItemComplexComponent*

A ComplexComponent specifically for generic structures.

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__(**kwargs)	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Return the capability of moving of an item.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new BoardItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	Returns False.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

`collides_with`(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

`debug_info`()

Return a string with the list of the attributes and their current value.

Return type
str

`detach`(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns False. A component is never pickable by itself (either the whole complex item is pickable or not, but not partially)

Example:

```
if item.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.17 GenericStructure

class pygamelib.board_items.**GenericStructure**(*value=0*, ***kwargs*)

Bases: *Immovable*

A GenericStructure is as the name suggest, a generic object to create all kind of structures.

It can be tweaked with all the properties of *BoardItem*, *Immovable* and it can be made pickable, overlappable or restorable or any combination of these.

If you need an action to be done when a Player and/or a NPC touch the structure please have a look at *pygamelib.board_items.GenericActionableStructure*.

Parameters

- **pickable** (*bool*) – Define if the structure can be picked-up by a Player or NPC.
- **overlappable** (*bool*) – Define if the structure can be overlapped by a Player or NPC.
- **restorable** (*bool*) – Define if the structure can be restored by the Board after a Player or NPC passed through. For example, you want a door or an activator structure (see GenericActionableStructure for that) to remain on the board after it's been overlapped by a player. But you could also want to develop some kind of Space Invaders game were the protection block are overlappable but not restorable.
- **value** (*int | float*) – The value of the structure. It can be used for scoring, resource spending, etc.

On top of these, this object takes all parameters of *BoardItem* and *Immovable*

Important: If you need a structure with a permission system please have a look at *GenericActionableStructure*. This class has a permission system for activation.

__init__(*value=0*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([value])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load data and create a new BoardItem out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

> **Important:** collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

`set_can_move`(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

`set_overlappable`(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

`set_pickable`(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

`set_restorable`(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

`property size`

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.18 Immovable

class pygamelib.board_items.**Immovable**(*inventory_space: int = None*, ***kwargs*)

Bases: *BoardItem*

This class derive *BoardItem* and describe an object that cannot move or be moved (like a wall). *can_move()* cannot be configured and return False. The other properties can be configured. They have the same default values than *BoardItem*.

Parameters

inventory_space (*int*) – The space the immovable item takes into an *Inventory* (in case the item is pickable). By default it is 0.

__init__(*inventory_space: int = None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__([inventory_space])	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Return the capability of moving of an item.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new BoardItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	Returns True if the item is pickable, False otherwise.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset:* Vector2D *= None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.19 Movable

`class pygamelib.board_items.Movable`(*step: int = None*, *step_vertical: int = None*, *step_horizontal: int = None*, *movement_speed: float = None*, ***kwargs*)

Bases: `BoardItem`

A class representing BoardItem capable of movements.

Movable subclasses `BoardItem`.

Parameters

- **step** (*int*) – the amount of cell a movable can cross in one turn. Default value: 1.
- **step_vertical** (*int*) – the amount of cell a movable can vertically cross in one turn. Default value: step value.
- **step_horizontal** (*int*) – the amount of cell a movable can horizontally cross in one turn. Default value: step value.
- **movement_speed** (*int | float*) – The time (in seconds) between 2 movements of a Movable. It is used by all the Game's actuation methods to enforce move speed of NPC and projectiles.

The movement_speed parameter is only used when the Game is configured with MODE_RT. Additionally the dtmove property is used to accumulate time between frames. It is entirely managed by the Game object and most of the time you shouldn't mess up with it. Unless you want to manage movements by yourself. If so, have fun! That's the point of the pygamelib to let you do whatever you like.

This class derive BoardItem and describe an object that can move or be moved (like a player or NPC). Thus this class implements BoardItem.can_move(). However it does not implement BoardItem.pickable() or BoardItem.overlappable()

`__init__`(*step: int = None*, *step_vertical: int = None*, *step_horizontal: int = None*, *movement_speed: float = None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([step, step_vertical, ...])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Movable implements can_move().
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`has_inventory`()	This is a virtual method that must be implemented in deriving class.
`load`(data)	Load data and create a new Movable out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Serialize the Immovable object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory() → bool

This is a virtual method that must be implemented in deriving class. This method has to return True or False. This represent the capacity for a Movable to have an inventory.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new Movable out of it.

Parameters

data (*dict*) – Data to create a new movable item (usually generated by *serialize()*)

Returns

A new complex item.

Return type

~pygamelib.board_items.Movable

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns

The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the Immovable object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.20 NPC

class pygamelib.board_items.**NPC**(*actuator=None*, ***kwargs*)

Bases: *Character*

A class that represent a non playable character controlled by the computer. For the NPC to be successfully managed by the Game, you need to set an actuator.

None of the parameters are mandatory, however it is advised to make good use of some of them (like type or name) for game design purpose.

In addition to its own member variables, this class inherits all members from:

- *pygamelib.board_items.Character*
- *pygamelib.board_items.Movable*
- *pygamelib.board_items.BoardItem*

This class sets a couple of variables to default values:

- max_hp: 10
- hp: 10
- remaining_lives: 1
- attack_power: 5
- **movement_speed: 0.25 (one movement every 0.25 second). Only useful if the game** mode is set to MODE_RT.

Parameters

actuator (pygamelib.actuators.Actuator) – An actuator, it can be any class but it need to implement pygamelib.actuators.Actuator.

Example:

```
mynpc = NPC(name='Idiot McStupid', type='dumb_enemy')
mynpc.step = 1
mynpc.actuator = RandomActuator()
```

__init__(*actuator=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__([actuator])	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Movable implements can_move().
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
has_inventory()	Define if the NPC has an inventory.
load(data)	Load data and create a new NPC out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	Define if the NPC is pickable.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Serialize the NPC object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset:* Vector2D = *None*)

Tells if this item collides with another item.

> **Important:** collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory()

Define if the NPC has an inventory.

This method returns false because the game engine doesn't manage NPC inventory yet but it could be in the future. It's a good habit to check the value returned by this function.

Returns
False

Return type
Boolean

Example:

```
if mynpc.has_inventory():
    print("Cool: we can pickpocket that NPC!")
else:
    print("No pickpocketing XP for us today :(")
```

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new NPC out of it.

Parameters
data (*dict*) – Data to create a new npc (usually generated by *serialize()*)

Returns
A new npc.

Return type
~pygamelib.board_items.NPC

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Define if the NPC is pickable.

Obviously this method always return False.

Returns
False

Return type
Boolean

Example:

```
if mynpc.pickable():
    Utils.warn("Something is fishy, that NPC is pickable"
        "but is not a Pokemon...")
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the NPC object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns

The size.

Return type

list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.21 Player

class pygamelib.board_items.**Player**(*inventory=None*, ***kwargs*)

Bases: `Character`

A class that represent a player controlled by a human.

This can take all parameter from `Character`, `Movable` and obviously `BoardItem`.

It is a specific board item as the whole Game class assumes only one player. Aside from the wrapper functions (like Game.move_player for example), there is no reel limitations to use more than one player.

The player also has a couple of attributes that are added for your convenience. You are free to use them or not. They are (name and default value):

- max_hp: 100
- hp: 100
- remaining_lives: 3
- attack_power: 10
- **movement_speed: 0.1 (one movement every 0.1 second). Only useful if the game mode**
  is set to MODE_RT.
- **inventory: A `Inventory` object. If none is provided,**
  one is created automatically.

A player can be animated by providing a `Animation` object to its *animation* attribute.

Like all other board items, you can specify a *sprixel* attribute that will be the representation of the player on the board.

Example:

```
player = Player(
    name="Player",
    # A sprixel with "@" as the model, no background color, a cyan foreground
    # color and we set the background to be transparent.
    sprixel=core.Sprixel("@", None, core.Color(0, 255, 255), True),
```

(continues on next page)

(continued from previous page)

```
    max_hp=200,
)
```

__init__(*inventory=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__([inventory])	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Movable implements can_move().
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
has_inventory()	This method returns True (a player has an inventory).
load(data)	Load data and create a new Character out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	Returns True if the item is overlappable, False otherwise.
pickable()	This method returns False (a player is obviously not pickable).
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	Returns True if the item is restorable, False otherwise.
serialize()	Serialize the Character object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
dtmove	
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event’s value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a “FQDN style” string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory()

This method returns True (a player has an inventory).

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new Character out of it.

Parameters
data (*dict*) – Data to create a new character item (usually generated by *serialize()*)

Returns
A new character item.

Return type
~pygamelib.board_items.Character

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

This method returns False (a player is obviously not pickable).

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the Character object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters

value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns

The size.

Return type

list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.22 Projectile

`class pygamelib.board_items.Projectile`(*name='projectile'*, *direction=Direction.RIGHT*, *step=1*, *range=5*, *model=''*, *movement_animation=None*, *hit_animation=None*, *hit_model=None*, *hit_callback=None*, *is_aoe=False*, *aoe_radius=0*, *parent=None*, *callback_parameters=None*, *movement_speed=0.15*, *collision_exclusions=None*, ***kwargs*)

Bases: `Movable`

A class representing a projectile type board item. That class can be sub-classed to represent all your needs (fireballs, blasters shots, etc.).

That class support the 2 types of representations: model and animations. The animation cases are slightly more evolved than the regular item.animation. It does use the item.animation but with more finesse as a projectile can travel in many directions. So it also keeps track of models and animation per travel direction.

You probably want to subclass Projectile. It is totally ok to use it as it, but it is easier to create a subclass that contains all your Projectile information and let the game engine deal with orientation, range keeping, etc. Please see examples/07_projectiles.py for a good old fireball example.

By default, Projectile travels in straight line in one direction. This behavior can be overwritten by setting a specific actuator (a projectile is a `Movable` so you can use my_projectile.actuator).

The general way to use it is as follow:

- Create a factory object with your static content (usually the static models, default direction and hit callback)
- Add the direction related models and/or animation (keep in mind that animation takes precedence over static models)
- deep copy that object when needed and add it to the projectiles stack of the game object.
- use Game.actuate_projectiles(level) to let the Game engine do the heavy lifting.

The Projectile constructor takes the following parameters:

Parameters

- **direction** (*Direction*) – A direction from the *constants* module
- **range** (*int*) – The maximum range of the projectile in number of cells that can be crossed. When range is attained the hit_callback is called with a BoardItemVoid as a collision object.
- **step** (*int*) – the amount of cells a projectile can cross in one turn
- **model** (*str*) – the default model of the projectile.
- **movement_animation** (*Animation*) – the default animation of a projectile. If a projectile is sent in a direction that has no explicit and specific animation, then movement_animation is used if defined.
- **hit_animation** (*Animation*) – the animation used when the projectile collide with something.
- **hit_model** (*str*) – the model used when the projectile collide with something.
- **hit_callback** (*function*) – A reference to a function that will be called upon collision. The hit_callback is receiving the object it collides with as first parameter.
- **is_aoe** (*bool*) – Is this an 'area of effect' type of projectile? Meaning, is it doing something to everything around (mass heal, exploding rocket, fireball, etc.)? If yes, you must set that parameter to True and set the aoe_radius. If not, the Game object will only send the colliding object in front of the projectile.

- **aoe_radius** (*int*) – the radius of the projectile area of effect. This will force the Game object to send a list of all objects in that radius.
- **callback_parameters** (*list*) – A list of parameters to pass to hit_callback.
- **movement_speed** (*int | float*) – The movement speed of the projectile
- **collision_exclusions** (*list*) – A list of **TYPES** of objects that should not collides with that projectile. It is usually a good idea to put the projectile type in the exclusion list. This prevent the projectile to collide with other instances of itself. Adding the projectile's emitter is also a valid idea.
- **parent** – The parent object (usually a Board object or some sort of BoardItem).

Important: The effects of a Projectile are determined by the callback. No callback == no effect!

Example:

```
fireball = Projectile(
                        name="fireball",
                        model=Utils.red_bright(black_circle),
                        hit_model=graphics.Models.EXPLOSION,
                        # won't collide with other projectiles.
                        collision_exclusions = [Projectile],
                    )
fireball.set_direction(Direction.RIGHT)
my_game.add_projectile(1, fireball,
                       my_game.player.pos[0], my_game.player.pos[1] + 1)
```

__init__(*name='projectile'*, *direction=Direction.RIGHT*, *step=1*, *range=5*, *model=''*, *movement_animation=None*, *hit_animation=None*, *hit_model=None*, *hit_callback=None*, *is_aoe=False*, *aoe_radius=0*, *parent=None*, *callback_parameters=None*, *movement_speed=0.15*, *collision_exclusions=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__([name, direction, step, range, ...])	Like the object class, this class constructor takes no parameter.
add_directional_animation(direction, animation)	Add an animation for a specific direction.
add_directional_model(direction, model)	Add an model for a specific direction.
attach(observer)	Attach an observer to this instance.
can_move()	Movable implements can_move().
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
directional_animation(direction)	Return the animation for a specific direction.
directional_model(direction)	Return the model for a specific direction.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.

continues on next page

Table 7 – continued from previous page

`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`has_inventory`()	Projectile cannot have inventory by default.
`hit`(objects)	A method that is called when the projectile hit something.
`load`(data)	Load data and create a new Movable out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Projectile are overlappable by default.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`remove_directional_animation`(direction)	Remove an animation for a specific direction.
`remove_directional_model`(direction)	Remove the model for a specific direction.
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	We assume that by default, Projectiles are restorable.
`serialize`()	Serialize the Immovable object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_direction`(direction)	Set the direction of a projectile
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

`animation`	A property to get and set an `Animation` for this item.
`column`	Convenience method to get the current stored column of the item.
`direction`	The direction of the projectile.
`dtmove`	
`heading`	Return the heading of the item.
`height`	Convenience method to get the height of the item.
`inventory_space`	A property to get and set the size that the BoardItem takes in the `Inventory`.
`layer`	Convenience method to get the current stored layer number of the item.
`model`	
`particle_emitter`	
`row`	Convenience method to get the current stored row of the item.
`screen_column`	A property to get/set the screen column.
`screen_row`	A property to get/set the screen row.
`size`	A read-only property that gives the size of the item as a 2 dimensions list.
`width`	Convenience method to get the width of the item.

add_directional_animation(*direction*, *animation*)

Add an animation for a specific direction.

Parameters

- **direction** (*int*) – A direction from the constants module.
- **animation** (*Animation*) – The animation for the direction

Example:

```
fireball.add_directional_animation(Direction.UP, Direction.UP, animation)
```

add_directional_model(*direction*, *model*)

Add an model for a specific direction.

Parameters

- **direction** (*int*) – A direction from the constants module.
- **model** (*str*) – The model for the direction

Example:

```
fireball.add_directional_animation(Direction.UP, upward_animation)
```

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move() → bool

Movable implements can_move().

Returns
True

Return type
Boolean

`collides_with`(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

`property column`

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

`debug_info`()

Return a string with the list of the attributes and their current value.

Return type
str

`detach`(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

property direction

The direction of the projectile.

Updating this property also updates the UnidirectionalActuator's direction.

Parameters
value (int | *Vector2D*) – some param

> **Warning:** If your projectile uses directional model and/or animation you should use *set_direction()* to set the projectile direction.

Example:

```
bullet.direction = Vector2D(0, 1)
```

directional_animation(*direction*)

Return the animation for a specific direction.

Parameters
direction (*int*) – A direction from the constants module.

Return type
Animation

Example:

```
# No more animation for the UP direction
fireball.directional_animation(Direction.UP)
```

directional_model(*direction*)

Return the model for a specific direction.

Parameters
direction (*int*) – A direction from the constants module.

Return type
str

Example:

```
fireball.directional_model(Direction.UP)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

property dtmove

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

has_inventory()

Projectile cannot have inventory by default.

Returns
False

Return type
Boolean

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

hit(*objects*)

A method that is called when the projectile hit something.

That method is automatically called by the Game object when the Projectile collide with another object or is at the end of its range.

Here are the call cases covered by the Game object:

- range is reached without collision and projectile IS NOT an AoE type: hit() is called with a single BoardItemVoid in the objects list.
- range is reached without collision and projectile IS an AoE type: hit() is called with the list of all objects within aoe_radius (including structures).
- projectile collide with something and IS NOT an AoE type: hit() is called with the single colliding object in the objects list.
- projectile collide with something and IS an AoE type: hit() is called with the list of all objects within aoe_radius (including structures).

In turn, that method calls the hit_callback with the following parameters (in that order):

1. the projectile object
2. the list of colliding objects (that may contain only one object)
3. the callback parameters (from the constructor callback_parameters)

Parameters
objects – A list of objects hit by or around the projectile.

Example:

```
my_projectile.hit([npc1])
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

`property layer`

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

`classmethod load`(*data*)

Load data and create a new Movable out of it.

Parameters
data (*dict*) – Data to create a new movable item (usually generated by *serialize()*)

Returns
A new complex item.

Return type
~pygamelib.board_items.Movable

`property model`

`notify`(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

`overlappable`()

Projectile are overlappable by default.

Returns
True

Return type
Boolean

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

remove_directional_animation(*direction*)

Remove an animation for a specific direction.

Parameters
direction (*int*) – A direction from the constants module.

Example:

```
# No more animation for the UP direction
fireball.remove_directional_animation(Direction.UP)
```

remove_directional_model(*direction*)

Remove the model for a specific direction.

Parameters
direction (*int*) – A direction from the constants module.

Example:

```
fireball.directional_model(Direction.UP)
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

We assume that by default, Projectiles are restorable.

Returns
True

Return type
bool

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Serialize the Immovable object.

This returns a dictionary that contains all the key/value pairs that makes up the object.

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_direction(*direction*)

Set the direction of a projectile

This method will set a UnidirectionalActuator with the direction. It will also take care of updating the model and animation for the given direction if they are specified.

Parameters
direction (*int*) – A direction from the constants module.

Example:

```
fireball.set_direction(Direction.UP)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
```

(continues on next page)

(continued from previous page)

```
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.23 TextItem

class pygamelib.board_items.**TextItem**(*text=None*, ***kwargs*)

Bases: *BoardComplexItem*

New in version 1.2.0.

The text item is a board item that can contains text. The text can then be manipulated and placed on a *Board*.

It is overall a *BoardComplexItem* (so it takes all the parameters of that class). The big difference is that the first parameter is the text you want to display.

The text parameter can be either a regular string or a *Text* object (in case you want formatting and colors).

Parameters

text (str | *Text*) – The text you want to display.

Example:

```
city_name = TextItem('Super City')
fancy_city_name = TextItem(text=base.Text('Super City', base.Fore.GREEN,
    base.Back.BLACK,
    base.Style.BRIGHT
))
my_board.place_item(city_name, 0, 0)
my_board.place_item(fancy_city_name, 1, 0)
```

__init__(*text=None*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([text])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Returns True if the item can move, False otherwise.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new TextItem out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	A property to get and set the size that the BoardItem takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
text	The text within the item.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Returns True if the item can move, False otherwise.

Example:

```
if board.item(4,5).can_move():
    print('The item can move')
```

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type

bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns

The column coordinate

Return type

int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type

str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters

other (*BoardItem*) – The item you want to calculate the distance to.

Returns

The distance between this item and the other.

Return type

float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

Warning: Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

A property to get and set the size that the BoardItem takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new TextItem out of it.

Parameters
data (*dict*) – Data to create a new text item (usually generated by *serialize()*)

Returns
A new complex npc.

Return type
~pygamelib.board_items.TextItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters
new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.

- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property text

The text within the item.

TextItem.text can be set to either a string or a *Text* object.

It will always return a *Text* object.

Internally it translate the text to a *Sprite* to display it correctly on a *Board*. If print()-ed it will do so like the *Text* object.

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.24 Tile

class pygamelib.board_items.**Tile**(***kwargs*)

Bases: *BoardComplexItem*, *GenericStructure*

New in version 1.2.0.

A Tile is a standard *BoardComplexItem* configured by default to:

- be overlappable
- be restorable
- be not pickable
- be immovable.

Aside from the movable attributes (it inherit from GenericStructure so it's an Immovable object), everything else is configurable.

It is particularly useful to display a *Sprite* on the background or to create terrain.

Example:

```
grass_sprite = Sprite.load_from_ansi_file('textures/grass.ans')
for pos in grass_positions:
    outdoor_level.place_item( Tile(sprite=grass_sprite), pos[0], pos[1] )
```

__init__(***kwargs*)

Parameters

- **overlappable** (*bool*) – Defines if the Tile can be overlapped.
- **restorable** (*bool*) – Defines is the Tile should be restored after being overlapped.
- **pickable** (*bool*) – Defines if the Tile can be picked up by the Player or NPC.

Please see *BoardComplexItem* for additional parameters.

Methods

`__init__`(**kwargs)	**param overlappable** Defines if the Tile can be overlapped.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	A Tile cannot move.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`item`(row, column)	Return the item component at the row, column position if it is within the complex item's boundaries.
`load`(data)	Load data and create a new Tile out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	Returns True if the item is overlappable, False otherwise.
`pickable`()	Returns True if the item is pickable, False otherwise.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the complex board item from the display buffer to the frame buffer.
`restorable`()	Returns True if the item is restorable, False otherwise.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.
`update_sprite`()	Update the complex item with the current sprite.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
sprite	A property to easily access and update a complex item's sprite.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

A Tile cannot move.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset:* Vector2D = *None*)

Tells if this item collides with another item.

> **Important:** collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

`display`()

Print the model WITHOUT carriage return.

`distance_to`(*other*)

Calculates the distance with an item.

Parameters
`other` (*`BoardItem`*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

`handle_notification`(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **`subject`** (*`PglBaseObject`*) – The object that has changed.
- **`attribute`** (*`str`*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **`value`** (*`Any`*) – The new value of the attribute. This can be None.

`property heading`

Return the heading of the item.

This is a read only property that is updated by *`store_position()`*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

item(*row*, *column*)

Return the item component at the row, column position if it is within the complex item's boundaries.

Return type
~pygamelib.board_items.BoardItem

Raises
PglOutOfBoardBoundException – if row or column are out of bound.

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```python
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new Tile out of it.

Parameters

data (*dict*) – Data to create a new tile (usually generated by *serialize()*)

Returns

A new complex npc.

Return type

~pygamelib.board_items.Tile

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```python
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

Returns True if the item is overlappable, False otherwise.

Example:

```python
if board.item(4,5).overlappable():
    print('The item is overlappable')
```

property particle_emitter

pickable()

Returns True if the item is pickable, False otherwise.

Example:

```python
if board.item(4,5).pickable():
    print('The item is pickable')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns

The position as a 2D vector

Return type
Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the complex board item from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

Returns True if the item is restorable, False otherwise.

Example:

```
if board.item(4,5).restorable():
    print('The item is restorable')
```

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

property sprite

A property to easily access and update a complex item's sprite.

Parameters

new_sprite (*Sprite*) – The sprite to set

Example:

```
npc1 = board_items.ComplexNpc(
                                sprite=npc_sprite_collection['npc1_idle']
                            )
# to access the sprite:
if npc1.sprite.width * npc1.sprite.height > CONSTANT_BIG_GUY:
    game.screen.place(
        base.Text(
            'Big boi detected!!!',
            core.Color(255,0,0),
            style=TextStyle.BOLD,
        ),
        notifications.row,
        notifications.column,
    )
# And to set it:
if game.player in game.neighbors(3, npc1):
    npc1.sprite = npc_sprite_collection['npc1_fight']
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

update_sprite()

Update the complex item with the current sprite.

Note: This method use to need to be called every time the sprite was changed. Starting with version 1.3.0, it is no longer a requirement as BoardComplexItem.sprite was turned into a property that takes care of calling update_sprite().

Example:

```
item = BoardComplexItem(sprite=position_idle)
for s in [walk_1, walk_2, walk_3, walk_4]:
    # This is not only no longer required but also wasteful as
    # update_sprite() is called twice here.
    item.sprite = s
    item.update_sprite()
    board.move(item, Direction.RIGHT, 1)
    time.sleep(0.2)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.25 Treasure

class pygamelib.board_items.**Treasure**(*value=10*, ***kwargs*)

Bases: *Immovable*

A Treasure is an *Immovable* that is pickable and with a non zero value. It is an helper class that allows to focus on game design and mechanics instead of small building blocks.

Parameters

- **model** (*str*) – The model that will represent the treasure on the map
- **value** (*int*) – The value of the treasure, it is usually used to calculate the score.

- **inventory_space** (*int*) – The space occupied by the treasure. It is used by *Inventory* as a measure of space. If the treasure's size exceed the Inventory size (or the cumulated size of all items + the treasure exceed the inventory max_size()) the Inventory will refuse to add the treasure.

Note: All the options from *Immovable* are also available to this constructor.

Example:

```
money_bag = Treasure(
    model=graphics.Models.MONEY_BAG,value=100,inventory_space=2
)
print(f"This is a money bag {money_bag}")
player.inventory.add_item(money_bag)
print(f"The inventory value is {player.inventory.value()} and is at
    {player.inventory.size()}/{player.inventory.max_size}")
```

__init__(*value=10*, ***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([value])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`can_move`()	Return the capability of moving of an item.
`collides_with`(other[, projection_offset])	Tells if this item collides with another item.
`debug_info`()	Return a string with the list of the attributes and their current value.
`detach`(observer)	Detach an observer from this instance.
`display`()	Print the model WITHOUT carriage return.
`distance_to`(other)	Calculates the distance with an item.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load data and create a new BoardItem out of it.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`overlappable`()	This represent the capacity for a Treasure to be overlapped by player or NPC.
`pickable`()	This represent the capacity for a Treasure to be picked-up by player or NPC.
`position_as_vector`()	Returns the current item position as a Vector2D
`render_to_buffer`(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
`restorable`()	This represent the capacity for a Treasure to be restored after being overlapped.
`serialize`()	Return a dictionary with all the attributes of this object.
`set_can_move`(value)	Set the value of the can_move property to value.
`set_overlappable`(value)	Set the value of the overlappable property to value.
`set_pickable`(value)	Set the value of the pickable property to value.
`set_restorable`(value)	Set the value of the restorable property to value.
`store_position`(row, column[, layer])	Store the BoardItem position for self access.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset: Vector2D = None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

This represent the capacity for a Treasure to be overlapped by player or NPC.

A treasure is not overlappable.

Returns
False

Return type
bool

property particle_emitter

pickable()

This represent the capacity for a Treasure to be picked-up by player or NPC.

A treasure is obviously pickable by the player and potentially NPCs. `Board` puts the Treasure in the `Inventory` if the picker implements has_inventory()

Returns
True

Return type
bool

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type
`Vector2D`

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

This represent the capacity for a Treasure to be restored after being overlapped.

A treasure is not overlappable, therefor is not restorable.

Returns
False

Return type
bool

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns
The row coordinate

Return type
int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

`serialize()` → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

`set_can_move`(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

`set_overlappable`(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

`set_pickable`(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

`set_restorable`(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

`property size`

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.4.26 Wall

class pygamelib.board_items.**Wall**(***kwargs*)

Bases: *Immovable*

A Wall is a specialized *Immovable* object that as unmodifiable characteristics:

- It is not pickable (and cannot be).
- It is not overlappable (and cannot be).
- It is not restorable (and cannot be).

As such it's an object that cannot be moved, cannot be picked up or modified by Player or NPC and block their ways. It is therefor advised to create one per board and reuse it in many places.

Parameters

- **model** (*str*) – The representation of the Wall on the Board.
- **name** (*str*) – The name of the Wall.
- **size** (*int*) – The size of the Wall. This parameter will probably be deprecated as size is only used for pickable objects.

__init__(***kwargs*)

Like the object class, this class constructor takes no parameter.

Methods

__init__(**kwargs)	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
can_move()	Return the capability of moving of an item.
collides_with(other[, projection_offset])	Tells if this item collides with another item.
debug_info()	Return a string with the list of the attributes and their current value.
detach(observer)	Detach an observer from this instance.
display()	Print the model WITHOUT carriage return.
distance_to(other)	Calculates the distance with an item.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load data and create a new BoardItem out of it.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
overlappable()	This represent the capacity for a BoardItem to be overlapped by player or NPC.
pickable()	This represent the capacity for a BoardItem to be pick-up by player or NPC.
position_as_vector()	Returns the current item position as a Vector2D
render_to_buffer(buffer, row, column, ...)	Render the board item into a display buffer (not a screen buffer).
restorable()	This represent the capacity for an Immovable Movable item.
serialize()	Return a dictionary with all the attributes of this object.
set_can_move(value)	Set the value of the can_move property to value.
set_overlappable(value)	Set the value of the overlappable property to value.
set_pickable(value)	Set the value of the pickable property to value.
set_restorable(value)	Set the value of the restorable property to value.
store_position(row, column[, layer])	Store the BoardItem position for self access.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

animation	A property to get and set an *Animation* for this item.
column	Convenience method to get the current stored column of the item.
heading	Return the heading of the item.
height	Convenience method to get the height of the item.
inventory_space	Return the size that the Immovable item takes in the *Inventory*.
layer	Convenience method to get the current stored layer number of the item.
model	
particle_emitter	
row	Convenience method to get the current stored row of the item.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size	A read-only property that gives the size of the item as a 2 dimensions list.
width	Convenience method to get the width of the item.

property animation

A property to get and set an *Animation* for this item.

Important: When an animation is set, the item is setting the animation's parent to itself.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

can_move()

Return the capability of moving of an item.

Obviously an Immovable item is not capable of moving. So that method always returns False.

Returns
False

Return type
bool

collides_with(*other*, *projection_offset:* Vector2D *= None*)

Tells if this item collides with another item.

Important: collides_with() does not take the layer into account! It is not desirable for the pygamelib to assume that 2 items on different layers wont collide. For example, if a player is over a door, they are on different layers, but logically speaking they are colliding. The player is overlapping the door. Therefor, it is the responsibility of the developer to check for layers in collision, if it is important to the game logic.

Parameters

- **other** (*BoardItem*) – The item you want to check for collision.
- **projection_offset** (*Vector2D*) – A vector to offset this board item's position (not the position of the *other* item). Use this to detect a collision before moving the board item. You can pass the movement vector before moving to check if a collision will occur when moving.

Return type
bool

Example:

```
if projectile.collides_with(game.player):
    game.player.hp -= 5
```

property column

Convenience method to get the current stored column of the item.

This is absolutely equivalent to access to item.pos[1].

Returns
The column coordinate

Return type
int

Example:

```
if item.column != item.pos[1]:
    print('Something extremely unlikely just happened...')
```

debug_info()

Return a string with the list of the attributes and their current value.

Return type
str

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display()

Print the model WITHOUT carriage return.

distance_to(*other*)

Calculates the distance with an item.

Parameters
other (*BoardItem*) – The item you want to calculate the distance to.

Returns
The distance between this item and the other.

Return type
float

Example:

```
if npc.distance_to(game.player) <= 2.0:
    npc.seek_and_destroy = True
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property heading

Return the heading of the item.

This is a read only property that is updated by *store_position()*.

The property represent the orientation and movement of the item in the board. It gives the difference between the item's centroid current and previous position. Thus, giving you both the direction and the distance of the movement. You can get the angle from here.

One of the possible usage of that property is to set the sprite/sprixel/model of a moving item.

Returns
The heading of the item.

Return type
Vector2D

Example:

```
if my_item.heading.column > 0:
    my_item.sprixel.model = item_models["heading_right"]
```

> **Warning:** Just after placing an item on the board, and before moving it, the heading cannot be trusted! The heading represent the direction and orientation of the **movement**, therefore, it is not reliable before the item moved.

property height

Convenience method to get the height of the item.

This is absolutely equivalent to access to item.size[1].

Returns
The height

Return type
int

Example:

```
if item.height > board.height:
    print('The item is too big for the board.')
```

property inventory_space

Return the size that the Immovable item takes in the *Inventory*.

Returns
The size of the item.

Return type
int

property layer

Convenience method to get the current stored layer number of the item.

This is absolutely equivalent to access to item.pos[2].

Returns
The layer number

Return type
int

Example:

```
if item.layer != item.pos[2]:
    print('Something extremely unlikely just happened...')
```

classmethod load(*data*)

Load data and create a new BoardItem out of it.

Parameters
data (*dict*) – Data to create a new item (usually generated by *serialize()*)

Returns
A new item.

Return type
~pygamelib.board_items.BoardItem

property model

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

overlappable()

This represent the capacity for a *BoardItem* to be overlapped by player or NPC.

Returns
False

Return type
bool

property particle_emitter

pickable()

This represent the capacity for a *BoardItem* to be pick-up by player or NPC.

Returns
False

Return type
bool

Example:

```
if mywall.pickable():
    print('Whoaa this wall is really light... and small...')
else:
    print('Really? Trying to pick-up a wall?')
```

position_as_vector()

Returns the current item position as a Vector2D

Returns
The position as a 2D vector

Return type

Vector2D

Example:

```
gravity = Vector2D(9.81, 0)
next_position = item.position_as_vector() + gravity.unit()
```

render_to_buffer(*buffer*, *row*, *column*, *height*, *width*)

Render the board item into a display buffer (not a screen buffer).

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

restorable()

This represent the capacity for an *Immovable* Movable item. A wall is not overlappable.

Returns

False

Return type

bool

property row

Convenience method to get the current stored row of the item.

This is absolutely equivalent to access to item.pos[0].

Returns

The row coordinate

Return type

int

Example:

```
if item.row != item.pos[0]:
    print('Something extremely unlikely just happened...')
```

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize() → dict

Return a dictionary with all the attributes of this object.

Returns
A dictionary with all the attributes of this object.

Return type
dict

set_can_move(*value*)

Set the value of the can_move property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_can_move(False)
```

set_overlappable(*value*)

Set the value of the overlappable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_overlappable(False)
```

set_pickable(*value*)

Set the value of the pickable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_pickable(False)
```

set_restorable(*value*)

Set the value of the restorable property to value.

Parameters
value (*bool*) – The value to set.

Example:

```
item.set_restorable(False)
```

property size

A read-only property that gives the size of the item as a 2 dimensions list. The first element is the width and the second the height.

Returns
The size.

Return type
list

Example:

```
# This is a silly example because the Board object does not allow
# that use case.
if item.column + item.size[0] >= board.width:
    Game.instance().screen.display_line(
        f"{item.name} cannot be placed at {item.pos}."
    )
```

store_position(*row: int*, *column: int*, *layer: int = 0*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self position. It is a redundant information and might not be synchronized.

Parameters

- **row** (*int*) – the row of the item in the *Board*.
- **column** (*int*) – the column of the item in the *Board*.
- **layer** – the layer of the item in the *Board*. By default layer is set to 0.

Example:

```
item.store_position(3,4)
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

Convenience method to get the width of the item.

This is absolutely equivalent to access to item.size[0].

Returns
The width

Return type
int

Example:

```
if item.width > board.width:
    print('The item is too big for the board.')
```

3.5 constants

Accessible constants are the following:

General purpose:

- PYGAMELIB_VERSION

enum `pygamelib.constants.`**`Alignment`**`(`*value*`)`

Alignment regroup constants that used for various alignment purpose when organizing UI elements or other such graphical elements.

V_CENTER and H_CENTER respectively stand for Vertical center and Horizontal Center.

Member Type
`int`

Valid values are as follows:

`LEFT = <Alignment.LEFT: 30000011>`

`RIGHT = <Alignment.RIGHT: 30000100>`

`CENTER = <Alignment.CENTER: 30000101>`

`TOP = <Alignment.TOP: 30000110>`

`BOTTOM = <Alignment.BOTTOM: 30000111>`

`V_CENTER = <Alignment.V_CENTER: 30001000>`

`H_CENTER = <Alignment.H_CENTER: 30001001>`

enum `pygamelib.constants.`**`Algorithm`**`(`*value*`)`

A set of constants to identify the different algorithms used in the library (when a choice is possible). For now, it's only the path finding algorithm.

Member Type
`int`

Valid values are as follows:

`BFS = <Algorithm.BFS: 90000100>`

`ASTAR = <Algorithm.ASTAR: 90000101>`

enum `pygamelib.constants.`**`Direction`**`(`*value*`)`

Direction hold the basic constants for directions in the pygamelib. It is used for a wide variety of use cases from moving a player or NPC to indicate the direction of the movement of a cursor in the UI module!

Member Type
`int`

Valid values are as follows:

`NO_DIR = <Direction.NO_DIR: 10000000>`

`UP = <Direction.UP: 10000001>`

`DOWN = <Direction.DOWN: 10000010>`

LEFT = <Direction.LEFT: 10000011>

RIGHT = <Direction.RIGHT: 10000100>

DRUP = <Direction.DRUP: 10000101>

DRDOWN = <Direction.DRDOWN: 10000110>

DLUP = <Direction.DLUP: 10000111>

DLDOWN = <Direction.DLDOWN: 10001000>

enum pygamelib.constants.**EngineConstant**(*value*)

A couple of constants that controls the behavior of the engine itself.

Member Type
int

Valid values are as follows:

NO_PLAYER = <EngineConstant.NO_PLAYER: 90000001>

enum pygamelib.constants.**EngineMode**(*value*)

Constants that controls the mode of the engine. So far, it's a choice between real time and turn by turn, but in the future there could be additional modes.

Member Type
int

Valid values are as follows:

MODE_REAL_TIME = <EngineMode.MODE_REAL_TIME: 90000002>

MODE_TURN_BY_TURN = <EngineMode.MODE_TURN_BY_TURN: 90000003>

enum pygamelib.constants.**InputValidator**(*value*)

InputValidators are used in the UI module to indicate what type of inputs are valid and/or accepted from the user.

Member Type
int

Valid values are as follows:

INTEGER_FILTER = <InputValidator.INTEGER_FILTER: 50000001>

PRINTABLE_FILTER = <InputValidator.PRINTABLE_FILTER: 50000002>

enum pygamelib.constants.**Orientation**(*value*)

Orientation regroup constants that are used to describe the orientation of graphical elements. The best example, is the BoxLayout: it can be organized vertically or horizontally.

Member Type
int

Valid values are as follows:

HORIZONTAL = <Orientation.HORIZONTAL: 30000001>

VERTICAL = <Orientation.VERTICAL: 30000010>

enum `pygamelib.constants.`**`Permission`**(*value*)

Permission constants to separate what objects are allowed to interact with others. Mostly used to separate NPCs from Players.

Member Type
`int`

Valid values are as follows:

`PLAYER_AUTHORIZED = <Permission.PLAYER_AUTHORIZED: 20000001>`

`NPC_AUTHORIZED = <Permission.NPC_AUTHORIZED: 20000010>`

`ALL_CHARACTERS_AUTHORIZED = <Permission.ALL_CHARACTERS_AUTHORIZED: 20000011>`

`ALL_MOVABLE_AUTHORIZED = <Permission.ALL_MOVABLE_AUTHORIZED: 20000100>`

`NONE_AUTHORIZED = <Permission.NONE_AUTHORIZED: 20000101>`

enum `pygamelib.constants.`**`SizeConstraint`**(*value*)

SizeConstraint regroup constants that are used in element which the size can vary depending on context.

Member Type
`int`

Valid values are as follows:

`DEFAULT_SIZE = <SizeConstraint.DEFAULT_SIZE: 60000001>`

`MINIMUM_SIZE = <SizeConstraint.MINIMUM_SIZE: 60000002>`

`MAXIMUM_SIZE = <SizeConstraint.MAXIMUM_SIZE: 60000003>`

`EXPAND = <SizeConstraint.EXPAND: 60000004>`

enum `pygamelib.constants.`**`State`**(*value*)

A set of constants that describe the internal state of something. For example the, state of the *`Game`* engine that are used to process or not some events.

Member Type
`int`

Valid values are as follows:

`RUNNING = <State.RUNNING: 40000001>`

`PAUSED = <State.PAUSED: 40000010>`

`STOPPED = <State.STOPPED: 40000011>`

enum `pygamelib.constants.`**`TextStyle`**(*value*)

TextStyling is used to format characters or text. It is mostly used by *`Text`*.

Member Type
`str`

Valid values are as follows:

`BOLD = <TextStyle.BOLD: '\x1b[1m'>`

```
UNDERLINE = <TextStyle.UNDERLINE: '\x1b[4m'>
```

The following constants are used in versions <= 1.3.0 and have been deprecated starting version 1.4.0.

Directions:

- NO_DIR: This one is used when no direction can be provided by an actuator (destination reached for a PathFinder for example)
- UP
- DOWN
- LEFT
- RIGHT
- DRUP : Diagonal right up
- DRDOWN : Diagonal right down
- DLUP : Diagonal Left up
- DLDOWN : Diagonal left down

Permissions:

- PLAYER_AUTHORIZED
- NPC_AUTHORIZED
- ALL_PLAYABLE_AUTHORIZED (deprecated in 1.2.0 in favor of ALL_CHARACTERS_AUTHORIZED)
- ALL_CHARACTERS_AUTHORIZED
- ALL_MOVABLE_AUTHORIZED
- NONE_AUTHORIZED

UI positions:

- ORIENTATION_HORIZONTAL
- ORIENTATION_VERTICAL
- ALIGN_LEFT
- ALIGN_RIGHT
- ALIGN_CENTER

Actions states (for Actuators for example):

- RUNNING
- PAUSED
- STOPPED

Accepted input (mainly used in pygamelib.gfx.ui for input dialogs): * INTEGER_FILTER * PRINTABLE_FILTER

Path Finding Algorithm Constants:

- ALGO_BFS
- ALGO_ASTAR

Text styling constants:

- BOLD

- UNDERLINE

Special constants:

- NO_PLAYER : That constant is used to tell the Game object not to manage the player.
- MODE_RT : Set the game object to Real Time mode. The game runs independently from the user input.
- MODE_TBT : Set the game object to Turn By Turn mode. The game runs turn by turn and pause between each user input.

3.6 engine

3.6.1 Board

`class pygamelib.engine.`**`Board`**`(`*name: str = 'Board', size: list = None, ui_borders: str = None, ui_border_bottom: str = '-', ui_border_top: str = '-', ui_border_left: str = '|', ui_border_right: str = '|', ui_board_void_cell=' ', ui_board_void_cell_sprixel:* Sprixel *= None, player_starting_position: list = None, DISPLAY_SIZE_WARNINGS=False, parent=None, partial_display_viewport=None, partial_display_focus=None, enable_partial_display=False*`)`

Bases: `PglBaseObject`

A class that represent a game board.

The board object is a 2D matrix of board items. This means that you can visualize it as a chessboard for example. All board items are positioned on this chessboard-like object and can be moved around.

The Board object is the base object to build a level. Once created to your liking you can add items from the *board_items* module. You can also derived `BoardItem` to create your own board items, specific to your game.

If you want a detailed introduction to the Board object, go the the pygamelib wiki and read the "Getting started: the Board" article.

Note: In version 1.3.0 a new screen rendering stack was introduced. With this came the need for some object to hold more information about their state. This is the case for Board. To use partial display with the `Screen` buffer system the board itself needs to hold the information about were to draw and on what to focus on. The existing code will still work as the `Game` object takes care of forwarding the information to the Board. However, it is now possible to exploit the `Camera` object to create cut scenes and more interesting movements.

Important: Partial display related parameters are information used by the `display_around()` method and the `Screen` object to either display directly the board (display_around) or render the Board in the frame buffer. **You have to make sure that the focus element's position is updated**. If you use the player, you have nothing to do but the Camera object needs to be manually updated for example.

Warning: in 1.3.0 the notion of layers was added to the Board object. Layers are used to better manage items overlapping. For the moment, layers are automatically managed to expand and shrink on demand (or on a need basis). You can use the layer system to add some depth to your game but you should be warned that you may experience some issues. If it is the case please report them on the Github issues page. For existing code, the entire Board object behaves exactly like in version 1.2.x.

__init__(*name: str = 'Board'*, *size: list = None*, *ui_borders: str = None*, *ui_border_bottom: str = '-'*, *ui_border_top: str = '-'*, *ui_border_left: str = '|'*, *ui_border_right: str = '|'*, *ui_board_void_cell=' '*, *ui_board_void_cell_sprixel:* Sprixel *= None*, *player_starting_position: list = None*, *DISPLAY_SIZE_WARNINGS=False*, *parent=None*, *partial_display_viewport=None*, *partial_display_focus=None*, *enable_partial_display=False*)

Parameters

- **name** (*str*) – the name of the Board
- **size** (*list*) – array [width,height] with width and height being int. The size of the board. If layers is not specified it is set to 5.
- **player_starting_position** (*list*) – array [row,column] with row and column being int. The coordinates at which Game will place the player on change_level().
- **ui_borders** (*str*) – To set all the borders to the same value
- **ui_border_left** (*str*) – A string that represents the left border.
- **ui_border_right** (*str*) – A string that represents the right border.
- **ui_border_top** (*str*) – A string that represents the top border.
- **ui_border_bottom** (*str*) – A string that represents the bottom border.
- **ui_board_void_cell** (*str*) – A string that represents an empty cell. This option is going to be the model of the BoardItemVoid (see *pygamelib.board_items.BoardItemVoid*)
- **parent** (*Game*) – The parent object (usually the Game object).
- **DISPLAY_SIZE_WARNINGS** (*bool*) – A boolean to show or hide the warning about boards bigger than 80 rows and/or columns.
- **enable_partial_display** (*bool*) – A boolean to tell the Board to enable or not partial display of boards. Default: False.
- **partial_display_viewport** (*list*) – A 2 int elements array that gives the **radius** of the partial display in number of row and column. Please see *display_around()*.
- **partial_display_focus** (*BoardItem* or *Vector2D*) – An item to focus (i.e center) the view on. When partial display is enabled the rendered view will be centered on this focus point/item. It can be an item or a vector.

Methods

`__init__`([name, size, ui_borders, ...])	**param name** the name of the Board
`attach`(observer)	Attach an observer to this instance.
`check_sanity`()	Check the board sanity.
`clear_cell`(row, column[, layer])	Clear cell (row, column, layer)
`detach`(observer)	Detach an observer from this instance.
`display`()	Display the entire board.
`display_around`(item, row_radius, column_radius)	Display only a part of the board.
`generate_void_cell`()	This method return a void cell.
`get_immovables`(**kwargs)	Return a list of all the Immovable objects in the Board.
`get_movables`(**kwargs)	Return a list of all the Movable objects in the Board.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`init_board`()	Initialize the board with BoardItemVoid that uses ui_board_void_cell_sprixel or ui_board_void_cell (in that order of preference) as model.
`init_cell`(row, column[, layer])	Initialize a specific cell of the board with BoardItemVoid that uses ui_board_void_cell as model.
`instantiate_item`(data)	Instantiate a BoardItem from its serialized data.
`item`(row, column[, layer])	Return the item at the row, column, layer position if within board's boundaries.
`layers`(row, column)	A method to get the number of layers at the Board's given coordinates.
`load`([data])	Create a new Board object based on serialized data.
`move`(item, direction[, step])	Board.move() is a routing function.
`neighbors`(obj[, radius])	Returns a list of neighbors (non void item) around an object.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`place_item`(item, row, column[, layer, ...])	Place an item at coordinates row, column and layer.
`remove_item`(item)	Remove an item from the board.
`render_cell`(row, column)	New in version 1.3.0.
`render_to_buffer`(buffer, row, column, ...)	Render the board into from the display buffer to the frame buffer.
`serialize`()	Return a serialized version of the board.
`store_screen_position`(row, column)	Store the screen position of the object.

Attributes

height	A convenience read only property to get the height of the Board.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
width	A convenience read only property to get the width of the Board.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

check_sanity() → None

Check the board sanity.

This is essentially an internal method called by the constructor.

clear_cell(*row*, *column*, *layer=0*)

Clear cell (row, column, layer)

This method clears a cell, meaning it position a void_cell BoardItemVoid at these coordinates.

It also removes the items from the the list of movables and immovables.

Parameters

- **row** (*int*) – The row of the item to remove
- **column** (*int*) – The column of the item to remove
- **layer** (*int*) – The layer of the item to remove. The default value is 0 to remain coherent with previous version of the library.

Example:

```
myboard.clear_cell(3,4,0)
```

Warning: This method does not check the content before, it *will* overwrite the content.

Important: In the case of a BoardComplexItem derivative (Tile, ComplexPlayer , ComplexNPC, etc.) clearing one cell of the entire item is enough to remove the entire item from the list of movables or immovables.

Note: Starting in 1.3.0 and the addition of board's layers, there is no more overlapping matrix. With no more moving items around this method should be a little faster. It also means that the layer parameter is really important (a wrong layer means that you'll clear the wrong cell). Be ready to catch an IndexError exception

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
`observer` (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display() → None

Display the entire board.

This method display the Board (as in print()), taking care of displaying the borders, and everything inside.

It uses the __str__ method of the item, which by default uses (in order) BoardItem.sprixel and (if no sprixel is defined) BoardItem.model. If you want to override this behavior you have to subclass BoardItem.

display_around(*item*, *row_radius*, *column_radius*) → None

Display only a part of the board.

This method behaves like display() but only display a part of the board around an item (usually the player). Example:

```
# This will display only a total of 30 cells vertically and
# 60 cells horizontally.
board.display_around(player, 15, 30)
```

Parameters

- **item** (*BoardItem*) – an item to center the view on (it has to be a subclass of BoardItem)
- **row_radius** (*int*) – The radius of display in number of rows showed. Remember that it is a radius not a diameter…
- **column_radius** (*int*) – The radius of display in number of columns showed. Remember that… Well, same thing.

It uses the same display algorithm than the regular display() method.

generate_void_cell()

This method return a void cell.

If ui_board_void_cell_sprixel is defined it uses it, otherwise use ui_board_void_cell to generate the void item.

Returns
A void board item

Return type
BoardItemVoid

Example:

```
board.generate_void_cell()
```

get_immovables(***kwargs*)

Return a list of all the Immovable objects in the Board.

See *pygamelib.board_items.Immovable* for more on
an Immovable object.

Parameters
****kwargs** – an optional dictionnary with keys matching Immovables class members and value being something **contained** in that member.

Returns
A list of Immovable items

Example:

```
for m in myboard.get_immovables():
    print(m.name)

# Get all the Immovable objects that type contains "wall"
    AND name contains fire
walls = myboard.get_immovables(type="wall",name="fire")
```

get_movables(***kwargs*)

Return a list of all the Movable objects in the Board.

See *pygamelib.board_items.Movable* for more on a Movable object.

Parameters
****kwargs** – an optional dictionnary with keys matching Movables class members and value being something contained in that member.

Returns
A list of Movable items

Example:

```
for m in myboard.get_movables():
    print(m.name)

# Get all the Movable objects that has a type that contains "foe"
foes = myboard.get_movables(type="foe")
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property height

A convenience read only property to get the height of the Board.

It is absolutely equivalent to access to board.size[1].

Returns
The height of the board.

Return type
int

Example:

```
if board.size[1] != board.height:
    print('Houston, we have a problem...')
```

init_board()

Initialize the board with BoardItemVoid that uses ui_board_void_cell_sprixel or ui_board_void_cell (in that order of preference) as model.

This method is automatically called by the Board's constructor.

Example:

```
myboard.init_board()
```

init_cell(*row*, *column*, *layer=0*) → None

Initialize a specific cell of the board with BoardItemVoid that uses ui_board_void_cell as model.

Parameters

- **row** (*int*) – the row coordinate.
- **column** (*int*) – the column coordinate.

Example:

```
myboard.init_cell(2,3,0)
```

static instantiate_item(*data: dict*)

Instantiate a BoardItem from its serialized data.

Parameters
data (*dict*) – The data to use to build the item.

Returns
an instance of a `BoardItem`.

Important: The actual object depends on the serialized data. It can be any derivative of BoardItem (even custom objects as long as they inherit from BoardItem) as long as they are importable by this class.

Example:

```
# First get some board item serialization data. For example:
data = super_duper_npc.serialize()
# Then instantiate a new one:
another_super_duper_npc = Board.instantiate_item(data)
```

`item`(*row*, *column*, *layer=-1*)

Return the item at the row, column, layer position if within board's boundaries.

Parameters

- **row** (*int*) – The row to probe.
- **column** (*int*) – The column to probe.
- **layer** (*int*) – The layer to probe (default: -1 i.e the top item).

Return type
pygamelib.board_items.BoardItem

Raises
`PglOutOfBoardBoundException` – if row, column or layer are out of bound.

`layers`(*row*, *column*) → int

A method to get the number of layers at the Board's given coordinates.

Returns
The number of layers of the board.

Return type
int

Example:

```
if board.layers(game.player.row, game.player.column) > 1:
    print('The player is stomping on something!')
```

`classmethod load`(*data: dict = None*)

Create a new Board object based on serialized data.

If data is None, None is returned.

If a color component is missing from data, it is set to 0 (see examples).

Raises an exception if the color components are not integer.

Parameters
data (*dict*) – Data loaded from JSON data (serialized).

Returns
Either a Board object or None if data where empty.

Return type
`Board` | NoneType

Raise
PglInvalidTypeException

Example:

```
# Loading from parsed JSON data
new_board = Board.load(json.load("board_lvl_01.json"))
```

move(*item*, *direction*, *step=1*)

Board.move() is a routing function. It does 2 things:

1 - If the direction is a *Vector2D*, round the
values to the nearest integer (as move works with entire board cells, i.e integers).

2 - route toward the right moving function depending if the item is complex or
not.

Move an item in the specified direction for a number of steps.

Parameters

- **item** (pygamelib.board_items.Movable) – an item to move (it has to be a subclass of Movable)
- **direction** (*Direction* or *Vector2D*) – a direction from *constants*
- **step** (*int*) – the number of steps to move the item.

If the number of steps is greater than the Board, the item will be move to the maximum possible position.

If the item is not a subclass of Movable, an PglObjectIsNotMovableException exception (see *pygamelib.base.PglObjectIsNotMovableException*).

Example:

```
board.move(player,Direction.UP,1)
```

Important: if the move is successful, an empty BoardItemVoid (see pygamelib.boards_item.BoardItemVoid) will be put at the departure position (unless the movable item is over an overlappable item). If the movable item is over an overlappable item, the overlapped item is restored.

Important: Also important: If the direction is a *Vector2D*, the values will be rounded to the nearest integer (as move works with entire board cells). It allows for movement accumulation before actually moving. The step parameter is not used in that case.

neighbors(*obj*, *radius: int = 1*)

Returns a list of neighbors (non void item) around an object.

This method returns a list of objects that are all around an object between the position of an object and all the cells at **radius**.

Parameters

- **radius** (*int*) – The radius in which non void item should be included
- **obj** (*BoardItem*) – The central object. The neighbors are calculated for that object.

Returns
A list of BoardItem. No BoardItemVoid is included.

Raises

PglInvalidTypeException – If radius is not an int.

Example:

```
for item in game.neighbors(npc, 2):
    print(f'{item.name} is around {npc.name} at coordinates '
        '({item.pos[0]},{item.pos[1]})')
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

place_item(*item*, *row: int*, *column: int*, *layer: int = 0*, *auto_layer: bool = True*)

Place an item at coordinates row, column and layer.

If row, column or layer are out of the board boundaries, a PglOutOfBoardBoundException is raised.

If the item is not a subclass of BoardItem, a PglInvalidTypeException

The observers are notified of a successful placement with the pygamelib.engine.Board.place_item:item_placed event. The item that was deleted is passed as the value of the event.

> **Warning:** Nothing prevents you from placing an object on top of another. Be sure to check that. This method will check for items that are both overlappable **and** restorable to save them, but that's the extend of it.

remove_item(*item*)

Remove an item from the board.

If the item is a single BoardItem, this method is absolutely equivalent to calling *clear_cell()*. If item is a derivative of BoardComplexItem, it is not as clear_cell() only clears a specific cell (that can be part of a complex item). This method actually remove the entire item and clears all its cells.

The observers are notified of a successful removal with the pygamelib.engine.Board.remove_item:item_removed event. The item that was deleted is passed as the value of the event.

Parameters

item (*BoardItem*) – The item to remove.

Example:

```
game.current_board().remove_item(game.player)
```

render_cell(*row*, *column*)

New in version 1.3.0.

Render the cell at given position.

This method always return a *Sprixel* (it could be an empty one though). It automatically render the highest item (if items are overlapping for example). If the rendered *Sprixel* is configured to have transparent background, this method is going to go through the layers to make sure that it is rendering the sprixels correctly (i.e: with the right background color).

For basic usage of the library it is unlikely that you will use it. It is part of the screen rendering stack introduced in version 1.3.0. Actually unless you need to write a different rendering system you won't use that method.

Parameters

- **row** (*int*) – The row to render.
- **column** (*int*) – The column to render.

Return type

Sprixel

Raises

PglOutOfBoardBoundException – if row or column are out of bound.

Example:

```
# This renders the board from the top left corner of the screen.
for row in range(0, myboard.height):
    for column in range(0, myboard.height):
        myscreen.place(
            myboard.render_cell(row, column)
        ),
        row,
        column,
```

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the board into from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A frame buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Return a serialized version of the board.

Returns
A dictionary containing the board's attributes.

Example:

```
serialized_board_data = myboard.serialize()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width

A convenience read only property to get the width of the Board.

It is absolutely equivalent to access to board.size[0].

Returns
The width of the board.

Return type
int

Example:

```
if board.size[0] != board.width:
    print('Houston, we have a problem...')
```

3.6.2 Game

class pygamelib.engine.**Game**(*name: str = 'Game', player:* Player *= None, boards={}, current_level=None, enable_partial_display=False, partial_display_viewport=None, partial_display_focus=None, mode=EngineMode.MODE_TURN_BY_TURN, user_update=None, input_lag=0.01, user_update_paused=None*)

Bases: `PglBaseObject`

A class that serve as a game engine.

This object is the central system that allow the management of a game. It holds boards (see `pygamelib.engine.Board`), associate it to level, takes care of level changing, etc.

Note: The game object has an object_library member that is always an empty array except just after loading a board. In this case, if the board have a "library" field, it is going to be used to populate object_library. This library is accessible through the Game object mainly so people have access to it across different Boards during level design in the editor. That architecture decision is debatable.

Note: The constructor of Game takes care of initializing the terminal to properly render the colors on Windows.

Important: The Game object automatically assumes ownership over the Player.

__init__(*name: str = 'Game', player:* Player *= None, boards={}, current_level=None, enable_partial_display=False, partial_display_viewport=None, partial_display_focus=None, mode=EngineMode.MODE_TURN_BY_TURN, user_update=None, input_lag=0.01, user_update_paused=None*)

Parameters

- **name** (*str*) – The Game name.
- **boards** (*dict*) – A dictionary of boards with the level number as key and a board reference as value.
- **current_level** (*int*) – The current level.
- **enable_partial_display** (*bool*) – A boolean to tell the Game object to enable or not partial display of boards. Default: False.
- **partial_display_viewport** (*list*) – A 2 int elements array that gives the **radius** of the partial display in number of row and column. Please see `display_around()`.
- **partial_display_focus** (*BoardItem*) – The object that is going to be the center of the view when the board is displayed.
- **mode** (*EngineMode*) – The mode parameter configures the way the run() method is going to behave. The default value is EngineMode.MODE_TURN_BY_TURN. In that mode, the Game object wait for an user input before looping. Exactly like when you wait for user input with get_key(). The other possible value is EngineMode.MODE_TURN_BY_TURN. RT stands for "Real Time". In that mode, the Game object waits for a minimal amount of time (0.01 i.e 100 FPS, configurable through the input_lag parameter) in order to get the input from the user and call the update function right away. This parameter is *only* useful if you use Game.run().

- **user_update** (*function*) – A reference to the main program update function. The update function is called for each new frame. It is called with 3 parameters: the game object, the user input (can be None) and the elapsed time since last frame.
- **user_update_paused** (*function*) – A reference to the update function called when the game is paused. It is called with the same 3 parameters than the regular update function: the game object, the user input (can be None) and the elapsed time since last frame. If not specified, the regular update function is called but nothing is done regarding NPCs, projectiles, animations, etc.
- **input_lag** (*float | int*) – The amount of time the run() function is going to wait for a user input before returning None and calling the update function. Default is 0.01.

Methods

`__init__`([name, player, boards, ...])	**param name** The Game name.
`actuate_npcs`(level_number[, elapsed_time])	Actuate all NPCs on a given level
`actuate_projectiles`(level_number[, elapsed_time])	Actuate all Projectiles on a given level
`add_board`(level_number, board)	Add a board for the level number.
`add_npc`(level_number, npc[, row, column, ...])	Add a NPC to the game.
`add_projectile`(level_number, projectile[, ...])	Add a Projectile to the game.
`animate_items`(level_number[, elapsed_time])	That method goes through all the BoardItems of a given map and call Animation.next_frame().
`attach`(observer)	Attach an observer to this instance.
`change_level`(level_number)	Change the current level, load the board and place the player to the right place.
`clear_screen`()	Clear the whole screen (i.e: remove everything written in terminal)
`clear_session_logs`()	Delete all the log lines from the logs.
`config`([section])	Get the content of a previously loaded configuration section.
`create_config`(section)	Initialize a new config section.
`current_board`()	This method return the board object corresponding to the current_level.
`delete_all_levels`()	Delete all boards and their associated levels from the game object.
`delete_level`([lvl_number])	Delete a level and its associated Board from the game object.
`detach`(observer)	Detach an observer from this instance.
`display_board`()	Display the current board.
`display_player_stats`([life_model, void_model])	Display the player name and health.
`get_board`(level_number)	This method returns the board associated with a level number.
`get_key`()	Reads the next key-stroke returning it as a string.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`insert_board`(level_number, board)	Insert a board for the level number.

continues on next page

Table 8 – continued from previous page

instance(*args, **kwargs)	Returns the instance of the Game object
load_board(filename[, lvl_number])	Load a saved board
load_config(filename[, section])	Load a configuration file from the disk.
move_player(direction[, step])	Easy wrapper for Board.move().
neighbors([radius, obj])	Get a list of neighbors (non void item) around an object.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
pause()	Set the game engine state to PAUSE.
remove_npc(level_number, npc)	This methods remove the NPC from the level in parameter.
run()	New in version 1.2.0.
save_board(lvl_number, filename)	Save a board to a JSON file
save_config([section, filename, append])	Save a configuration section.
session_log(line)	Add a line to the session logs.
session_logs()	Return the complete session logs since instantiation.
start()	Set the game engine state to RUNNING.
stop()	Set the game engine state to STOPPED.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
state	Get/set the state of the game.

actuate_npcs(*level_number*, *elapsed_time=0.0*)

Actuate all NPCs on a given level

This method actuate all NPCs on a board associated with a level. At the moment it means moving the NPCs but as the Actuators become more capable this method will evolve to allow more choice (like attack use objects, etc.)

When all NPCs have been successfully actuated, the observers are notified of the change with the pygamelib.engine.Game.actuate_npcs:npcs_actuated event. Their is value passed for that event.

Parameters

- **level_number** (*int*) – The number of the level to actuate NPCs in.
- **elapsed_time** (*float*) – The amount of time that passed since last call. This parameter is not mandatory.

Example:

```
mygame.actuate_npcs(1)
```

Note: This method only move NPCs when their actuator state is RUNNING. If it is PAUSED or STOPPED, the NPC is not moved.

Note: Since version 1.2.0 it's possible for a Movable item to have different vertical and horizontal movement steps, so actuate_npc respect that by integrating the steps with a unit direction vector. It should be completely transparent and you should not expect any change. Just more movement freedom. If you do experience issues, please report a bug.

Note: Since version 1.2.0 and the appearance of the realtime mode, we have to account for movement speed. This method does it.

actuate_projectiles(*level_number*, *elapsed_time=0.0*)

Actuate all Projectiles on a given level

This method actuate all Projectiles on a board associated with a level. This method differs from actuate_npcs() as some logic is involved with projectiles that NPC do not have. This method decrease the available range by projectile.step each time it's called. It also detects potential collisions. If the available range falls to 0 or a collision is detected the projectile hit_callback is called.

This method respects the Projectile.collision_exclusions parameter and does not register collisions with objects of a type present in that list.

Important: In this method, projectiles do not collide with overlappable items. If you want to detect collisions with overlappable objects, please implement your own projectile actuation method.

Parameters

- **level_number** (*int*) – The number of the level to actuate Projectiles in.
- **elapsed_time** (*float*) – The amount of time that passed since last call. This parameter is not mandatory.

When all Projectiles have been successfully actuated, the observers are notified of the change with the pygamelib.engine.Game.actuate_projectiles:projectiles_actuated event. Their is value passed for that event.

Example:

```
mygame.actuate_projectiles(1)
```

Note: This method only move Projectiles when their actuator state is RUNNING. If it is PAUSED or STOPPED, the Projectile is not moved.

Important: Please have a look at the *pygamelib.board_items.Projectile.hit()* method for more information on the projectile hit mechanic.

add_board(*level_number: int*, *board:* Board) → None

Add a board for the level number.

This method associate a Board (*pygamelib.engine.Board*) to a level number.

If the partial display is enabled at Game level (i.e: partial_display_viewport is not None and enable_partial_display is True), this method propagate the settings to the board automatically. Same for partial_display_focus.

Example:

```
game.add_board(1,myboard)
```

Parameters

- **level_number** (*int*) – the level number to associate the board to.
- **board** (`pygamelib.engine.Board`) – a Board object corresponding to the level number.

Raises

PglInvalidTypeException – If either of these parameters are not of the correct type.

add_npc(*level_number*, *npc*, *row=None*, *column=None*, *layer=None*, *auto_layer=True*)

Add a NPC to the game. It will be placed on the board corresponding to the level_number. If row and column are not None, the NPC is placed at these coordinates. Else, it's randomly placed in an empty cell.

Example:

```
game.add_npc(1,my_evil_npc,5,2)
```

Parameters

- **level_number** (*int*) – the level number of the board.
- **npc** (`pygamelib.board_items.NPC`) – the NPC to place.
- **row** (*int*) – the row coordinate to place the NPC at.
- **column** (*int*) – the column coordinate to place the NPC at.

If either of these parameters are not of the correct type, a PglInvalidTypeException exception is raised.

Important: If the NPC does not have an actuator, this method is going to affect a pygamelib.actuators.RandomActuator() to npc.actuator. And if npc.step == None, this method sets it to 1

add_projectile(*level_number*, *projectile*, *row=None*, *column=None*)

Add a Projectile to the game. It will be placed on the board corresponding to level_number. Neither row nor column can be None.

Example:

```
game.add_projectile(1, fireball, 5, 2)
```

Parameters

- **level_number** (*int*) – the level number of the board.
- **projectile** (*Projectile*) – the Projectile to place.
- **row** (*int*) – the row coordinate to place the Projectile at.
- **column** (*int*) – the column coordinate to place the Projectile at.

If either of these parameters are not of the correct type, a PglInvalidTypeException exception is raised.

Important: If the Projectile does not have an actuator, this method is going to affect pygamelib.actuators.RandomActuator(moveset=[RIGHT]) to projectile.actuator. And if projectile.step == None, this method sets it to 1.

animate_items(*level_number*, *elapsed_time=0.0*)

That method goes through all the BoardItems of a given map and call Animation.next_frame().

When all items have been successfully animated, the observers are notified of the change with the pygamelib.engine.Game.animate_items:items_animated event. Their is value passed for that event.

Parameters

- **level_number** (*int*) – The number of the level to animate items in.
- **elapsed_time** (*float*) – The amount of time that passed since last call. This parameter is not mandatory.

Raise

PglInvalidLevelException *PglInvalidTypeException*

Example:

```
mygame.animate_items(1)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

change_level(*level_number: int*) → None

Change the current level, load the board and place the player to the right place.

Example:

```
game.change_level(1)
```

Parameters

level_number (*int*) – the level number to change to.

Raises

base.PglInvalidTypeException – If parameter is not an int.

clear_screen()

Clear the whole screen (i.e: remove everything written in terminal)

Deprecated since version 1.2.0: Starting 1.2.0 we are using the pygamelib.engine.Screen object to manage the screen. That function is a simple forward and is kept for backward compatibility only. You should use Game.screen.clear()

clear_session_logs() → None

Delete all the log lines from the logs.

Example:

```
game = Game.instance()
game.clear_logs()
```

Note: The session log system is nothing more than a list to do your “debug prints”. If you want a real logging system, please use Python logging module.

config(*section: str = 'main'*) → dict

Get the content of a previously loaded configuration section.

Parameters
section (*str*) – The name of the section.

Example:

```
if mygame.config('main')['pgl-version-required'] < 10200:
    print('The pygamelib version 1.2.0 or greater is required.')
    exit()
```

create_config(*section: str*) → None

Initialize a new config section.

The new section is a dictionary.

Parameters
section (*str*) – The name of the new section.

Example:

```
if mygame.config('high_scores') is None:
    mygame.create_config('high_scores')
mygame.config('high_scores')['first_place'] = mygame.player.name
```

current_board() → *Board*

This method return the board object corresponding to the current_level.

Example:

```
game.current_board().display()
```

If current_level is set to a value with no corresponding board a PglException exception is raised with an invalid_level error.

delete_all_levels()

Delete all boards and their associated levels from the game object.

You might want to think twice before using that function…

Example:

```
game.delete_all_levels()
```

delete_level(*lvl_number: int = None*)

Delete a level and its associated Board from the game object.

Both the level and the board can't be used after that (unless they are reloaded or replaced of course).

Parameters
lvl_number (*int*) – The number of the level to remove.

Raises

- *base.PglInvalidTypeException* – If parameter is not an int.
- *base.PglInvalidLevelException* – If parameter is not a valid level.

Example:

```
my_game.delete_level(1)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display_board()

Display the current board.

The behavior of that function is dependant on how you configured this object. If you set enable_partial_display to True AND partial_display_viewport is set to a correct value, it will call Game.current_board().display_around() with the correct parameters. The partial display will be centered on the player (Game.player). Otherwise it will just call Game.current_board().display().

If the player is not set or is set to EngineConstant.NO_PLAYER partial display won't activate automatically.

Example:

```
mygame.enable_partial_display = True
# Number of rows, number of column (on each side, total viewport
# will be 20x20 in that case).
mygame.partial_display_viewport = [10, 10]
```

(continues on next page)

(continued from previous page)

```
# This will call Game.current_board().display_around()
mygame.display()
mygame.enable_partial_display = False
# This will call Game.current_board().display()
mygame.display()
```

display_player_stats(*life_model='\x1b[41m \x1b[0m'*, *void_model='\x1b[40m \x1b[0m'*)

Display the player name and health.

Deprecated since version This: method is completely deprecated and not even compatible with the Screen Buffer system. **It will be removed in 1.4.0**.

This method print the Player name, a health bar (20 blocks of life_model). When life is missing the complement (20-life missing) is printed using void_model. It also display the inventory value as "Score".

Parameters

- **life_model** (*str*) – The character(s) that should be used to represent the *remaining* life.
- **void_model** (*str*) – The character(s) that should be used to represent the *lost* life.

Note: This method might change in the future. Particularly it could take a template of what to display.

get_board(*level_number: int*) → *Board*

This method returns the board associated with a level number. :param level_number: The number of the level. :type level_number: int

Raises

PglInvalidTypeException – if the level_number is not an int.

Example:

```
level1_board = mygame.get_board(1)
```

static get_key()

Reads the next key-stroke returning it as a string.

Example:

```
key = Utils.get_key()
if key == Utils.key.UP:
    print("Up")
elif key == "q"
    exit()
```

Note: See *readkey* documentation in *readchar* package.

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

insert_board(*level_number: int*, *board:* Board) → None

Insert a board for the level number.

This method does basically the same thing than *add_board()* except that if the level number is already associated it re-affect the numbers down.

Example:

```
game.insert_board(1,myboard_1)
# level number 1 is associated with myboard_1
game.insert_board(2,myboard_2)
# level number 1 is associated with myboard_1
# level number 2 is associated with myboard_2
game.insert_board(2,myboard_3)
# level number 1 is associated with myboard_1
# level number 2 is now associated with myboard_3
# level number 3 is associated with myboard_2
```

Parameters

- **level_number** (*int*) – the level number to associate the board to.
- **board** (pygamelib.engine.Board) – a Board object corresponding to the level number.

Raises

PglInvalidTypeException – If either of these parameters are not of the correct type.

classmethod instance(**args*, ***kwargs*)

Returns the instance of the Game object

Creates a Game object on first call an then returns the same instance on further calls

Returns

Instance of Game object

load_board(*filename*, *lvl_number=0*)

Load a saved board

Load a Board saved on the disk as a JSON file. This method creates a new Board object, populate it with all the elements (except a Player) and then return it.

If the filename argument is not an existing file, the open function is going to raise an exception.

This method, load the board from the JSON file, populate it with all BoardItem included, check for sanity, init the board with BoardItemVoid and then associate the freshly created board to a lvl_number. It then create the NPCs and add them to the board.

Parameters

- **filename** (*str*) – The file to load

- **lvl_number** (*int*) – The level number to associate the board to. Default is 0.

Returns
a newly created board (see *pygamelib.engine.Board*)

Example:

```
mynewboard = game.load_board( 'awesome_level.json', 1 )
game.change_level( 1 )
```

load_config(*filename: str*, *section: str = 'main'*) → dict

Load a configuration file from the disk. The configuration file must respect the INI syntax. The goal of these methods is to simplify configuration files management.

Parameters

- **filename** (*str*) – The filename to load. does not check for existence.
- **section** (*str*) – The section to put the read config file into. This allow for multiple files for multiple purpose. Section is a human readable unique identifier.

Raises

- **FileNotFoundError** – If filename is not found on the disk.
- **json.decoder.JSONDecodeError** – If filename could not be decoded as JSON.

Returns
The parsed data.

Return type
dict

> **Warning: breaking changes:** before v1.1.0 that method use to load file using the configparser module. This have been dumped in favor of json files. Since that methods was apparently not used, there is no backward compatibility.

Example:

```
mygame.load_config('game_controls.json','game_control')
```

move_player(*direction*, *step=1*)

Easy wrapper for Board.move().

Example:

```
mygame.move_player(Direction.RIGHT,1)
```

neighbors(*radius=1*, *obj=None*)

Get a list of neighbors (non void item) around an object.

This method returns a list of objects that are all around an object between the position of an object and all the cells at **radius**.

Parameters

- **radius** (*int*) – The radius in which non void item should be included
- **object** (pygamelib.board_items.BoardItem) – The central object. The neighbors are calculated for that object. If None, the player is the object.

Returns
A list of BoardItem. No BoardItemVoid is included.

Raises
PglInvalidTypeException – If radius is not an int.

Example:

```
for item in game.neighbors(2):
    print(f'{item.name} is around player at coordinates '
        '({item.pos[0]},{item.pos[1]})')
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

pause()

Set the game engine state to PAUSE.

Example:

```
mygame.pause()
```

remove_npc(*level_number*, *npc*)

This methods remove the NPC from the level in parameter.

Parameters

- **level** (*int*) – The number of the level from where the NPC is to be removed.
- **npc** (*NPC*) – The NPC object to remove.

Example:

```
mygame.remove_npc(1, dead_npc)
```

run()

New in version 1.2.0.

The run() method act as the main game loop and does a number of things for you:

1. It grabs the user input. If the Game object is configured with MODE_TBT (the default), nothing happen until the user hit a key. If the mode is set to MODE_RT, it will wait for input_lag secondes for a user input before going to step 3.

2. It calculate the elapsed time between 2 frames.
3. Accumulates the elapsed time in the player dtmove variable (if there is a player object configured)
4. It sets the cursor position to 0,0 (meaning that your user_update function will draw on top of the previously drawn window). The Board.display() and Board.display_around() method clean the end of their line.
5. It calls the user_update function with 3 parameters: the game object, the key hit by the user (it can be None) and the elapsed time between to calls.
6. Clears the end of the screen.
7. Actuates NPCs (If there is at least one Board manage by Game).
8. Actuates projectiles (If there is at least one Board manage by Game).
9. Animates items (If there is at least one Board manage by Game).

On the subject of particle emitters, the `Board` object automatically update the ones that are attached to BoardItems. For all other particle emitters you need to call the update method of the emitters yourself (for now).

In version 1.2.X, there was a bug when the game was paused. In that case nothing was happening anymore. The user update function was not called and events were not processed. On top of that it was impossible to use run() without associating a board object with a level. Starting with version 1.3.0, it is now possible to use run() without associating a board object with a level. There is also a new parameter to the constructor (user_update_paused) that allows you to specify a function that will be called when the game is paused. This function will be called with the same 3 parameters than the regular update function: the game object, the user input (can be None) and the elapsed time since last frame.

Important: If you try to set the game state to PAUSED and the user_update_paused function is not defined, a notification will be issued and the game will continue to run. The notification message is pygamelib.engine.Game.run:PauseNotAvailable

Raises

PglInvalidTypeException, PglInvalidTypeException

Example:

```
mygame.run()
```

save_board(*lvl_number*, *filename*)

Save a board to a JSON file

This method saves a Board and everything in it but the BoardItemVoid.

Not check are done on the filename, if anything happen you get the exceptions from open().

Parameters

- **lvl_number** (*int*) – The level number to get the board from.
- **filename** (*str*) – The path to the file to save the data to.

Raises

- *PglInvalidTypeException* – If any parameter is not of the right type
- *PglInvalidLevelException* – If the level is not associated with a Board.

Example:

```
game.save_board( 1, 'hac-maps/level1.json')
```

If Game.object_library is not an empty array, it will be saved also.

> **Warning:** In version 1.3.0 the *Board* class changed a lot and a layer system has been added. Therefor, boards saved from version 1.3.0+ are *not* compatible with previous version. Previous boards can be loaded (*Game.load_board()* is backward compatible), but when saved they will be converted to the new format.

save_config(*section: str = None*, *filename: str = None*, *append: bool = False*) → None

Save a configuration section.

Parameters

- **section** (*str*) – The name of the section to save on disk.
- **filename** (*str*) – The file to write in. If not provided it will write in the file that was used to load the given section. If section was not loaded from a file, save will raise an exception.
- **append** (*bool*) – Do we need to append to the file or replace the content (True = append, False = replace)

Example:

```
mygame.save_config('game_controls', 'data/game_controls.json')
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

session_log(*line: str*) → None

Add a line to the session logs.

Session logs needs to be activated first.

Parameters
line (*str*) – The line to add to the logs.

Example:

```
game = Game.instance()
game.ENABLE_SESSION_LOGS = True
game.session_log('Game engine initialized')
```

Note: The session log system is nothing more than a list to do your “debug prints”. If you want a real logging system, please use Python logging module.

session_logs() → list

Return the complete session logs since instantiation.

Example:

```
game = Game.instance()
game.ENABLE_SESSION_LOGS = True
for line in game.logs():
    print(line)
```

Note: The session log system is nothing more than a list to do your “debug prints”. If you want a real logging system, please use Python logging module.

start()

Set the game engine state to RUNNING.

The game has to be RUNNING for actuate_npcs() and move_player() to do anything.

Example:

```
mygame.start()
```

property state

Get/set the state of the game.

Parameters
`value` (*State*) – The new state of the game (from the constants module).

Returns
The state of the game.

Return type
State

The observers are notified of a change of state with the pygamelib.engine.Game.state event. The new state is passed as the value of the event.

stop()

Set the game engine state to STOPPED.

Example:

```
mygame.stop()
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.6.3 Inventory

class pygamelib.engine.**Inventory**(*max_size=10*, *parent=None*)

Bases: `PglBaseObject`

A class that represent the Player (or NPC) inventory.

This class is pretty straightforward: it is an object container, you can add, get and remove items and you can get a value from the objects in the inventory.

On top of that, starting with version 1.3.0, a constraints system has been added. It allows to specify a certain amount of constraints that will be applied to the items when they are added to the inventory.

For the moment, constraints are limited to the number of items with a given type/ name/value (any combination of these three).

When a constraint is violated, the item is not added to the inventory and a notification is broadcasted to the observers of the inventory. A PglInventoryException is also raised with name “constraint_violation” and the constraint details in description.

Note: You can print() the inventory. This is mostly useful for debug as you want to have a better display in your game.

Warning: The `Game` engine and `Player` takes care to initiate an inventory for the player, you don’t need to do it.

__init__(*max_size=10*, *parent=None*)

The constructor takes two parameters: the maximum size of the inventory. And the Inventory owner/parent.

Each `BoardItem` that is going to be put in the inventory has a size (default is 1), the total addition of all these size cannot exceed max_size.

Parameters

- **max_size** (*int*) – The maximum size of the inventory. Default value: 10.
- **parent** – The parent object (usually a BoardItem).

Methods

__init__([max_size, parent])	The constructor takes two parameters: the maximum size of the inventory.
add_constraint(constraint_name[, item_type, ...])	
add_item(item)	Add an item to the inventory.
attach(observer)	Attach an observer to this instance.
available_space()	Return the available space in the inventory.
clear_constraints()	Remove all constraints from the inventory.
delete_item(name)	Delete THE FIRST item matching the name given in argument.
delete_items(name)	Delete ALL items matching the name given in argument.
detach(observer)	Detach an observer from this instance.
empty()	Empty the inventory.
get_item(name)	Return the FIRST item corresponding to the name given in argument.
get_items(name)	Return ALL items matching the name given in argument.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
items_name()	Return the list of all items names in the inventory.
load(data)	Load serialized data into a new Inventory object.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
remove_constraint(constraint_name)	
search(query)	Search for objects in the inventory.
serialize()	Serialize the inventory in a dictionary.
size()	Return the cumulated size of the inventory.
store_screen_position(row, column)	Store the screen position of the object.
value()	Return the cumulated value of the inventory.

Attributes

constraints	
items	Return the list of all items in the inventory.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

add_constraint(*constraint_name: str*, *item_type: str = None*, *item_name: str = None*, *item_value: int = None*, *max_number: int = 1*)

Add a constraint to the inventory.

Parameters

- **constraint_name** (*str*) – the name of the constraint.
- **item_type** (*str*) – the type of the item.
- **item_name** (*int*) – the name of the item.
- **item_value** – the value of the item.

- **max_number** (*int*) – the maximum number of items that match the item_* parameters that can be in the inventory.

The observers are notified of the addition of the constraint with the pygamelib.engine.Inventory.add_constraint event. The constraint that was added is passed as the value of the event as a dictionnary.

New in version 1.3.0.

add_item(*item*)

Add an item to the inventory.

This method will add an item to the inventory unless:

- it is not an instance of *BoardItem*,
- you try to add an item that is not pickable,
- there is no more space left in the inventory (i.e: the cumulated size of the inventory + your item.inventory_space is greater than the inventory max_size)
- An existing constraint is violated.

Parameters
item (*BoardItem*) – the item you want to add

Returns
The index of the newly added item in the inventory or None if the item could not be added.

Return type
int|None

Raise
PglInventoryException, *PglInvalidTypeException*

When an item is successfully added, the observers are notified of the change with the pygamelib.engine.Inventory.add_item event. The item that was added is passed as the value of the event.

When something goes wrong exceptions are raised. The following exceptions can be raised (*PglInventoryException*):

- not_pickable: The item you try to add is not pickable.
- not_enough_space: There is not enough space left in the inventory.
- constraint_violation: A constraint is violated.

A *PglInvalidTypeException* is raised when the item you try to add is not a *BoardItem*.

Example:

```
item = Treasure(model=graphics.Models.MONEY_BAG,size=2,name='Money bag')
try:
    mygame.player.inventory.add_item(item)
expect PglInventoryException as e:
    if e.error == 'not_enough_space':
        print(f"Impossible to add {item.name} to the inventory, there is no"
        "space left in it!")
        print(e.message)
    elif e.error == 'not_pickable':
        print(e.message)
```

Note: In versions prior to 1.3.0, the inventory object was changing the name of the item if another item with the same name was already in the inventory. This is (fortunately) not the case anymore. The Inventory class does NOT modify the items that are stored into it anymore.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

available_space() → int

Return the available space in the inventory.

That is to say, Inventory.max_size - Inventory.size().

The returned number is comprised between 0 and Inventory.max_size.

Returns
The size as an int.

Return type
int

Example:

```
method()
```

clear_constraints()

Remove all constraints from the inventory.

The observers are notified with the pygamelib.engine.Inventory.clear_constraints event. The value is set to None for this event.

New in version 1.3.0.

property constraints

Return the list of all constraints in the inventory.

Returns
a list of constraints (dict)

Return type
list

Example:

```
for cstr in game.player.inventory.constraints:
    print(f" - {cstr[name]}")
```

delete_item(*name*)

Delete THE FIRST item matching the name given in argument.

Parameters
name (*str*) – the name of the items you want to delete.

When an item is successfully removed, the observers are notified of the change with the pygamelib.engine.Inventory.delete_item event. The item that was deleted is passed as the value of the event.

Example:

```
mygame.player.inventory.delete_item('heart_1')
```

Important: Starting with version 1.3.0 this method does not raise exceptions anymore. It's behavior also changed from deleting a precise item to deleting the first one that matches the name.

delete_items(*name*)

Delete ALL items matching the name given in argument.

Parameters
name (*str*) – the name of the items you want to delete.

The observers are notified of each deletion with the pygamelib.engine.Inventory.delete_item event. The item that was deleted is passed as the value of the event.

Example:

```
mygame.player.inventory.delete_items('heart_1')
```

New in version 1.3.0.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

empty()

Empty the inventory.

The observers are notified that the Inventory has been emptied with the pygamelib.engine.Inventory.empty event. Nothing is passed as the value.

Example:

```
if inventory.size() > 0:
    inventory.empty()
```

get_item(*name*)

Return the FIRST item corresponding to the name given in argument.

Parameters
name (*str*) – the name of the item you want to get.

Returns
An item.

Return type
BoardItem | None

Example:

```
life_container = mygame.player.inventory.get_item('heart_1')
if isinstance(life_container,GenericActionableStructure):
    life_container.action(life_container.action_parameters)
```

Note: Please note that the item object reference is returned but nothing is changed in the inventory. The item hasn't been removed.

Important: Starting with version 1.3.0 this method does not raise exceptions anymore. Instead it returns None if no item is found. It's behavior also changed from returning a precise item to the first one that matches the name.

get_items(*name*)

Return ALL items matching the name given in argument.

Parameters
name (*str*) – the name of the item you want to get.

Returns
An array of items.

Return type
list

Example:

```
for life_container in mygame.player.inventory.get_items('heart_1'):
    if isinstance(life_container,GenericActionableStructure):
        life_container.action(life_container.action_parameters)
```

Note: Please note that the item object reference is returned but nothing is changed in the inventory. The item hasn't been removed.

New in version 1.3.0.

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property items

Return the list of all items in the inventory.

Returns
a list of *BoardItem*

Return type
list

Example:

```
for item in game.player.inventory.items:
    print(f"This is a mighty item: {item.name}")
```

items_name()

Return the list of all items names in the inventory.

Returns
a list of string representing the items names.

Return type
list

classmethod load(*data: dict*)

Load serialized data into a new Inventory object.

Parameters
data (*dict*) – The serialized data

Returns
A new Inventory object.

Return type
Inventory

New in version 1.3.0.

Example:

```
my_player.inventory = Inventory.load(data)
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

remove_constraint(*constraint_name: str*)

Remove a constraint from the inventory.

Parameters
constraint_name (*str*) – the name of the constraint.

The observers are notified of the removal of the constraint with the pygamelib.engine.Inventory.remove_constraint event. The constraint that was removed is passed as the value of the event as a dictionnary.

New in version 1.3.0.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

search(*query*)

Search for objects in the inventory.

All objects that matches the query are going to be returned. Search is performed on the name and type of the object.

Parameters
query – the query that items in the inventory have to match to be returned

Returns
a list of BoardItems.

Return type
list

Example:

```
for item in game.player.inventory.search('mighty'):
    print(f"This is a mighty item: {item.name}")
```

serialize()

Serialize the inventory in a dictionary.

Returns
The serialized data.

Return type
dict

New in version 1.3.0.

Example:

```
json.dump(my_inventory.serialize(), out_file)
```

size()

Return the cumulated size of the inventory. It can be used in the UI to display the size compared to max_size for example.

Returns
size of inventory

Return type
int

Example:

```
print(f"Inventory: {mygame.player.inventory.size()}/"
"{mygame.player.inventory.max_size}")
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

value()

Return the cumulated value of the inventory. It can be used for scoring for example.

Returns
value of inventory

Return type
int

Example:

```
if inventory.value() >= 10:
    print('Victory!')
    break
```

3.6.4 Screen

class pygamelib.engine.**Screen**(*width: int = None*, *height: int = None*)

Bases: *PglBaseObject*

The screen object is pretty straightforward: it is an object that allow manipulation of the screen.

> **Warning:** Starting with version 1.3.0 the terminal parameter has been removed. The Screen object now takes advantage of base.Console.instance() to get a reference to a blessed.Terminal object.

Version 1.3.0 introduced a new way of managing the screen. It rely on an internally managed display buffer that allows for easier positioning and more regular rendering. This comes at a cost though as the performances takes a hit. The screen should still be able to be refreshed between 50 and 60+ times per seconds (and still around 30 times per second within a virtual machine). These numbers obviously depends on the terminal used, the screen size and the content to display.

This change introduce two ways of displaying things on the screen:

- The **Improved Screen Management** stack (referred to as ISM later in the doc).
- The **Legacy Direct Display** stack.

It is safer to consider them mutually incompatible. In reality the **Improved Screen Management** will always use the whole display but you can use the methods from the **Direct Display** stack to write over the buffer. It is really **NOT** advised.

We introduced the **Improved Screen Management** stack because the direct display is messy and does not allow us to do what we want in term of positioning, UI, etc.

A typical usage consist of:

- Placing elements on the screen with *place()*
- Update the screen with *update()*

That's it! The screen maintain its own state and knows when to re-render the display buffer. You don't need to manually call *render()*. This helps with performances as the frame buffer is only rendered when needed.

Example:

```
screen = Screen()
# The next 3 lines do the same thing: display a message centered on the screen.
# Screen Buffer style
screen.place('This is centered', screen.vcenter, screen.hcenter)
screen.update()
# Direct Display style
screen.display_at('This is centered', screen.vcenter, screen.hcenter)
# The rest of this example uses the Screen Buffer (because placing a Board
# anywhere on the Screen is not supported by the Direct Display stack).
# delete the previous message and place a Board at the center of the screen
```

(continues on next page)

(continued from previous page)

```
screen.delete(screen.vcenter, screen.hcenter)
screen.place(
    my_awesome_board,
    screen.vcenter - int(my_awesome_board.height/2),
    screen.hcenter - int(my_awesome_board.width/2)
)
screen.update()
```

Precisions about the Improved Screen Management stack:

You don't need to know how the frame buffer works to use it. However, if you are interested in more details, here they are.

The Improved Screen Management stacks uses a double numpy buffer to represent the screen. One buffer is used to place elements as objects (that's the buffer managed by *place()* or *delete()*). It is never directly printed to the screen. It is here to simplify screen maintenance. This buffer is called the **display buffer**. It is practical to use to place, move and delete elements on the screen space. But as said before it cannot be directly printed to the screen. It needs to be rendered first.

For example, if you want to use a sprite on a title screen and want to move it around (or animate the screen). Normally (i.e with Direct Display) you would display the sprite at a specific position and then would either call *clear()* or overwrite all the sprite with spaces to erase and replace and/or move it. And that's very slow.

With the **Improved Screen Management** you *place()* the sprite and then just *delete()* it. And since it is only one object reference it is a very fast operation (we only place or delete one cell of the buffer).

When *update()* is called, it first look at the state of the buffers and call *render()* if needed (i.e: if something has change in the display buffer). The buffers are only rendered when needed.

When *render()* is called it goes through the display buffer and render each elements transforming it into a printable sequence that is stored in the frame buffer. The rendering is done from the bottom right corner of the screen to the top left corner. This allows for cleaning junk characters at no additional cost.

TL;DR: The **display buffer** hold the objects placed on the screen while the **frame buffer** hold the rendered representation of the display buffer.

The Screen object also inherits from the *PglBaseObject* and if the object that is *place()*-ed is an instance of *PglBaseObject*, the screen will automatically attach itself to the object. When notified of a change it will trigger a render cycle before the next update.

In terms of performances, depending on your terminal emulator and CPU you will most certainly achieve over 30 FPS. Here are a couple of benchmark results:

- On an Intel Core i7 @ 4.20 GHz: 50 to 70 FPS.
- On an AMD Ryzen 9 5950X @ 4.80 GHz: 60 to 100 FPS.

The new **Improved Screen Management** is faster than the legacy stack in most of the cases. The only case when the legacy Direct Display stack might be faster is in the case of a game or application with only simple ASCII characters and not a lot of things to display.

Here are some compiled benchmark results of both of systems over 150 runs:

Benchmark	Improved Screen Management	Legacy Direct Display
Sprite (place, render and update screen), Sprite size: 155x29	10.0 msec. or 71 FPS	380.0 msec. or 3 FPS
Sprite 200 updates	620.0 msec. or 76 FPS	9830.0 msec. or 20 FPS
Phase 1 - 500 frames. Single board avg load	11.02 msec. per frame or 91 FPS	12.65 msec. per frame or 79 FPS
Phase 2 - 500 frames. Dual board high load	18.18 msec. per frame or 55 FPS	28.34 msec. per frame or 35 FPS
Overall - 1000 frames.	14.60 msec. per frame or 68 FPS	20.49 msec. per frame or 49 FPS

You can use the 2 benchmark scripts to compare on your system:

- benchmark-screen-buffer.py
- benchmark-screen-direct-display.py

The frame buffer system has been tested on the following terminals:

- xterm-256color
- Konsole
- Kitty
- Alacritty
- GNOME Terminal

Performances are consistent across the different terminals. The only exception is the GNOME Terminal, which is slower than the others (about 20~30 % slower).

__init__(*width: int = None*, *height: int = None*)

The constructor takes the following (optional) parameters.

Parameters

- **width** (*int*) – The width of the screen.
- **height** (*int*) – The height of the screen.

Setting any of these parameters fixes the screen size regardless of the actual console/terminal resolution. Leaving any of these parameters unset will let the constructor use the actual console/terminal resolution instead.

Please have a look at the examples for more on this topic.

Example:

```
# Let's assume a terminal resolution of 170(width)x75(height).
screen = Screen()
# Next line display: "Screen width=170 height=75"
print(f"Screen width={screen.width} height={screen.height}")
screen = Screen(50)
# Next line display: "Screen width=50 height=75"
print(f"Screen width={screen.width} height={screen.height}")
screen = Screen(height=50)
# Next line display: "Screen width=170 height=50"
```

(continues on next page)

(continued from previous page)

```
print(f"Screen width={screen.width} height={screen.height}")
screen = Screen(50, 50)
# Next line display: "Screen width=50 height=50"
print(f"Screen width={screen.width} height={screen.height}")
```

Methods

__init__([width, height])	The constructor takes the following (optional) parameters.
attach(observer)	Attach an observer to this instance.
clear()	This methods clear the screen.
clear_buffers()	This methods clear the Screen's buffers (both display and frame buffer).
clear_frame_buffer()	This methods clear the frame buffer (but not the display buffer).
delete([row, column])	Delete a element on screen.
detach(observer)	Detach an observer from this instance.
display_at(text[, row, column, clear_eol, ...])	Displays text at a given position.
display_line(*text[, end, file, flush])	A wrapper to Python's print() builtin function except it will always add an ANSI sequence to clear the end of the line.
display_sprite(sprite[, filler, file, flush])	Displays a sprite at the current cursor position.
display_sprite_at(sprite[, row, column, ...])	Displays a sprite at a given position.
force_render()	Force the immediate rendering of the display buffer.
force_update()	Same as *force_render()* but also force the immediate screen update.
get(row, column)	Get an element from the display buffer at the specified screen coordinates.
handle_notification(subject[, attribute, value])	When a Screen object is notified, it set the display buffer to be rendered before the next update.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
place([element, row, column, rendering_pass])	Place an element on the screen.
render()	Render the display buffer into the frame buffer.
store_screen_position(row, column)	Store the screen position of the object.
trigger_rendering()	Trigger the frame buffer for rendering at the next update.
update()	Update the screen.

Attributes

buffer	The buffer property return a numpy.array as a writable frame buffer.
hcenter	Return the horizontal center of the screen as an int.
height	This property returns the height of the terminal window in number of characters.
need_rendering	This property return True if the display buffer has been updated since the last rendering cycle and the screen needs to re-render the frame buffer.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
vcenter	Return the vertical center of the screen as an int.
width	This property returns the width of the terminal window in number of characters.

`attach`(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property buffer

The buffer property return a numpy.array as a writable frame buffer.

The buffer is a 2D plane (like a screen) and anything can render in it. However, it is recommended to place objects through Screen.place() and update the screen with Screen.update() (update calls render() if needed and do the actual display).

Warning: Everything that is stored in the buffer *must* be printable. Each cell of the frame buffer represent a single character on screen, so you need to take care of that when you write into that buffer or you will corrupt the display. If *need_rendering* returns True, you need to manually call *render()* before writing anything into the frame buffer. Or else it will be squashed in the next rendering cycle.

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

`clear()`

This methods clear the screen.

`clear_buffers()`

This methods clear the Screen's buffers (both display and frame buffer).

Make sure that you really want to clear the buffers before doing so, because this is a slow operation.

Once the buffer is cleared nothing is left in it, you have to reposition (place) everything.

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

`clear_frame_buffer()`

This methods clear the frame buffer (but not the display buffer). This means that the next time *`update()`* is called, rendering will be triggered.

Make sure that you really want to clear the buffers before doing so, because this is a slow operation. It might however be faster than manually update screen cells.

Once the buffer is cleared nothing is left in it, it sets the Screen for a rendering update.

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

`delete`(*row=None*, *column=None*)

Delete a element on screen.

It is important to note that if you placed an element that occupies more than 1 cell, you only have to erase that specific position not the entire area.

Parameters

- **row** (*int*) – The row coordinate of the element to delete.
- **column** (*int*) – The column coordinate of the element to delete.

Example:

```
board = Board(size=[20,20])
screen.place(board, 2, 2)
# With this we have placed a board at screen coordinates 2,2 and the board
# will display on screen coordinates from 2,2 to 22,22.
# However, to delete the board we don't need to clean all these cells.
# Just the one where we placed the board:
screen.delete(2, 2)
```

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

display_at(*text*, *row=0*, *column=0*, *clear_eol=False*, *end='\n'*, *file=<colorama.ansitowin32.StreamWrapper object>*, *flush=False*)

Displays text at a given position. If clear_eol is True, also clear the end of line. Additionally you can specify all the parameters of a regular print() if you need to.

Parameters

- **text** (*str*) – The text to display. Please note that in that case text is a single string.
- **row** (*int*) – The row position in the terminal window.
- **column** (*int*) – The column position in the terminal window.
- **clear_eol** (*bool*) – If True this clears the end of the line (everything after the last character displayed by that method).
- **end** (*str*) – end sub string added to the printed text. Usually a carriage return.
- **file** (*stream*) –
- **flush** (*bool*) –

Important: The cursor is only moved for printing the text. It is returned to its previous position after.

Note: The position respect the row/column convention accross the library. It is reversed compared to the blessed module.

Example:

```
screen.display_at('This is centered',
                  int(screen.height/2),
                  int(screen.width/2),
                  clear_eol=True,
```

(continues on next page)

(continued from previous page)

```
                  end=''
                )
```

Note: This method is part of the **Legacy Direct Display** rendering stack and is incompatible with the methods identified as being part of the **Improved Screen Management** stack.

`display_line`(**text*, *end='\n'*, *file=<colorama.ansitowin32.StreamWrapper object>*, *flush=False*)

A wrapper to Python's print() builtin function except it will always add an ANSI sequence to clear the end of the line. Making it more suitable to use in a user_update callback.

The reason is that with line with variating length, if you use run() but not clear(), some characters will remain on screen because run(), for performances concerns does not clear the entire screen. It just bring the cursor back to the top left corner of the screen. So if you want to benefit from the increase performances you should use display_line().

Parameters

- ***text** (*str|objects*) – objects that can serialize to str. The ANSI sequence to clear the end of the line is *always* appended to the the text.
- **end** (*str*) – end sub string added to the printed text. Usually a carriage return.
- **file** (*stream*) –
- **flush** (*bool*) –

Example:

```
screen.display_line(f'This line will display correctly: {elapsed_time}')
# That line will have trailing characters that are not cleared after redraw
# if you don't use clear().
print(f'That one won't: {elapsed_time}')
```

New in version 1.2.0.

Note: This method is part of the **Legacy Direct Display** rendering stack and is incompatible with the methods identified as being part of the **Improved Screen Management** stack.

`display_sprite`(*sprite*, *filler= [0m*, *file=<colorama.ansitowin32.StreamWrapper object>*, *flush=False*)

Displays a sprite at the current cursor position. If a `Sprixel` is empty, then it's going to be replaced by filler.

Parameters

- **sprite** (`Sprite`) – The sprite object to display.
- **filler** (`Sprixel`) – A sprixel object to replace all empty sprixels in sprite.
- **file** (*stream*) –
- **flush** – print() parameter to flush the stream after printing

Examples:

```
screen.display_sprite(panda_sprite)
```

New in version 1.3.0.

Note: This method is part of the **Legacy Direct Display** rendering stack and is incompatible with the methods identified as being part of the **Improved Screen Management** stack.

display_sprite_at(*sprite*, *row=0*, *column=0*, *filler=* *[0m*, *file=<colorama.ansitowin32.StreamWrapper object>*, *flush=False*)

Displays a sprite at a given position. If a *Sprixel* is empty, then it's going to be replaced by filler.

Parameters

- **sprite** (*Sprite*) – The sprite object to display.
- **row** (*int*) – The row position in the terminal window.
- **column** (*int*) – The column position in the terminal window.
- **filler** (*Sprixel*) – A sprixel object to replace all empty sprixels in sprite.
- **file** (*stream*) –
- **flush** (*bool*) – print() parameter to flush the stream after printing

Example:

```
screen.display_sprite_at(panda_sprite,
                         int(screen.height/2),
                         int(screen.width/2)
                         )
```

New in version 1.3.0.

Note: This method is part of the **Legacy Direct Display** rendering stack and is incompatible with the methods identified as being part of the **Improved Screen Management** stack.

force_render()

Force the immediate rendering of the display buffer.

If you just want to mark the frame buffer for rendering before the next update use *trigger_rendering()* instead.

Example:

```
screen.force_render()
```

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

force_update()

Same as *force_render()* but also force the immediate screen update.

Example:

```
screen.force_update()
```

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

get(*row: int*, *column: int*)

Get an element from the display buffer at the specified screen coordinates.

The element is returned from the display buffer (pre-rendering).

Parameters

- **row** (*int*) – The row of the element to get.
- **column** (*int*) – The column of the element to get.

Example:

```
board = Board(size=[20,20])
screen.place(board, 2, 2)
my_board = screen.get(2,2)
```

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

handle_notification(*subject*, *attribute=None*, *value=None*)

When a Screen object is notified, it set the display buffer to be rendered before the next update.

property hcenter

Return the horizontal center of the screen as an int.

Example:

```
screen.place('horizontally centered', 0, screen.hcenter)
```

property height

This property returns the height of the terminal window in number of characters.

property need_rendering

This property return True if the display buffer has been updated since the last rendering cycle and the screen needs to re-render the frame buffer.

It returns False otherwise.

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

place(*element=None*, *row=None*, *column=None*, *rendering_pass=1*)

Place an element on the screen.

This method places an element in the screen display buffer. The element is then going to be rendered in the frame buffer before being printed on screen.

The following elements can be placed on screen:

- All BoardItem derivatives.
- All BoardComplexItem derivatives.
- *Board* object.
- *Text* objects.
- *Sprite* objects.
- *Sprixel* objects.
- Regular Python str.
- Any object that expose a render_to_buffer() method.

Here is the required signature for render_to_buffer:

render_to_buffer(self, buffer, row, column, buffer_height, buffer_width)

The buffer parameter will always be a numpy array, row and column are the position to render to. Finally buffer_height and buffer_width are the dimension of the buffer.

The buffer is rendered in 2 passes. By default all elements are rendered in pass 1. But if for some reason something needs to be drawn over other elements (like if a dialog/popup is needed for example), the element can be set to be rendered only during the second pass.

Parameters

- **element** (*various*) – The element to place.
- **row** (*int*) – The row to render to.
- **column** (*int*) – The column to render to.
- **rendering_pass** (*int*) – When to render the element. You can have any number of rendering passes but you have to be careful of performances. Higher passses render on top of lower passes. You can see the render passes as plane to write on. The default pass is 1.

> **Warning:** to be rendered on the second+ pass an element *needs* to implement render_to_buffer(…). This excludes all standard types (but not *Text*). Regular Python strings and object that can be print() can still be used in the first pass.

Example:

```
screen.place(my_sprite, 0, 0)
```

New in version 1.3.0.

> **Note:** This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

render()

Render the display buffer into the frame buffer.

Example:

```
screen.render()
screen.update()
```

New in version 1.3.0.

> **Note:** This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

trigger_rendering()

Trigger the frame buffer for rendering at the next update.

Example:

```
screen.trigger_rendering()
```

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Direct Display** stack.

update()

Update the screen. Update means write the frame buffer on screen.

Example:

```
mygame = Game()
sc = core.SpriteCollection.load_json_file('title_screens.spr')
mygame.screen.place(sc['welcome_screen'], 0, 0)
mygame.screen.update()
```

New in version 1.3.0.

Note: This method is part of the **Improved Screen Management** rendering stack and is incompatible with the methods identified as being part of the **Legacy Direct Display** stack.

property vcenter

Return the vertical center of the screen as an int.

Example:

```
screen.place('vertically centered', screen.vcenter, 0)
```

property width

This property returns the width of the terminal window in number of characters.

3.7 gfx

The gfx (for graphics) sub-module holds all the classes related to the graphics system.

3.7.1 core

Animation

class pygamelib.gfx.core.**Animation**(*display_time=0.05*, *auto_replay=True*, *frames=None*, *animated_object=None*, *refresh_screen=None*, *initial_index=None*, *parent=None*)

Bases: `object`

The Animation class is used to give the ability to have more than one model for a BoardItem. A BoardItem can have an animation and all of them that are available to the Game object can be animated through Game.animate_items(lvl_number). To benefit from that, BoardItem.animation must be set explicitely. An animation is controlled via the same state system than the Actuators.

The frames are all stored in a list called frames, that you can access through Animation.frames.

Parameters

- **display_time** (*float*) – The time each frame is displayed
- **auto_replay** (*bool*) – controls the auto replay of the animation, if false once the animation is played it stays on the last frame of the animation.
- **frames** (array[str| *Sprixel* | *Sprite*] | *SpriteCollection*) – an array of “frames” (string, sprixel, sprite) or a sprite collection
- **animated_object** (*BoardItem*) – The object to animate. This parameter is deprecated. Please use parent instead. It is only kept for backward compatibility. The parent parameter always takes precedence over this one.
- **parent** (*BoardItem*) – The parent object. It is also the object to animate. Important: We cannot animate anything else that BoardItems and subclasses.
- **refresh_screen** (*function*) – The callback function that controls the redrawing of the screen. This function reference should come from the main game.

Important: When a *SpriteCollection* is used as the *frames* parameter the sprites’ names are ordered so the frames are displayed in correct order. This means that ‘walk_1’ is going to be displayed before ‘walk_2’. Otherwise SpriteCollection is un-ordered.

Example

```python
def redraw_screen(game_object):
    game_object.clear_screen()
    game_object.display_board()

item = BoardItem(model=Sprite.ALIEN, name='Friendly Alien')
# By default BoardItem does not have any animation, we have to
# explicitly create one
item.animation = Animation(display_time=0.1, parent=item,
                           refresh_screen=redraw_screen)
```

__init__(*display_time=0.05*, *auto_replay=True*, *frames=None*, *animated_object=None*, *refresh_screen=None*, *initial_index=None*, *parent=None*)

Methods

__init__([display_time, auto_replay, ...])	
add_frame(frame)	Add a frame to the animation.
current_frame()	Return the current frame.
load(data)	Load a serialized Animation object.
next_frame()	Update the parent's model, sprixel or sprite with the next frame of the animation.
pause()	Set the animation state to PAUSED.
play_all()	Play the entire animation once.
remove_frame(index)	Remove a frame from the animation.
reset()	Reset the Animation to the first frame.
search_frame(frame)	Search a frame in the animation.
serialize()	Serialize the Animation object.
start()	Set the animation state to State.RUNNING.
stop()	Set the animation state to STOPPED.

Attributes

dtanimate	The time elapsed since the last frame was displayed.

add_frame(*frame*)

Add a frame to the animation.

The frame has to be a string (that includes sprites from the Sprite module and squares from the Utils module).

Raise an exception if frame is not a string.

Parameters

frame (str|:class:*Sprite `*|*:class: `Sprixel*) – The frame to add to the animation.

Raise

pygamelib.base.PglInvalidTypeException

Example:

```
item.animation.add_frame(Sprite.ALIEN)
item.animation.add_frame(Sprite.ALIEN_MONSTER)
```

current_frame()

Return the current frame.

Example:

```
item.model = item.animation.current_frame()
```

property dtanimate

The time elapsed since the last frame was displayed.

classmethod load(*data*)

Load a serialized Animation object.

Parameters
: **data** (*dict*) – The serialized Animation object.

Returns
: The loaded Animation object.

Return type
: *Animation*

next_frame()

Update the parent's model, sprixel or sprite with the next frame of the animation.

That method takes care of automatically resetting the animation if the last frame is reached if the state is State.RUNNING.

If the the state is PAUSED it still update the parent.model and returning the current frame. It does NOT actually go to next frame.

If parent is not a sub class of *BoardItem* an exception is raised.

Raise
: *PglInvalidTypeException*

Example:

```
item.animation.next_frame()
```

> **Warning:** If you use Sprites as frames, you need to make sure your Animation is attached to a *BoardComplexItem*.

pause()

Set the animation state to PAUSED.

Example:

```
item.animation.pause()
```

play_all()

Play the entire animation once.

That method plays the entire animation only once, there is no auto replay as it blocks the game (for the moment).

If the the state is PAUSED or STOPPED, the animation does not play and the method return False.

If parent is not a sub class of *BoardItem* an exception is raised.

If screen_refresh is not defined or is not a function an exception is raised.

Raise
: *PglInvalidTypeException*

Example:

```
item.animation.play_all()
```

remove_frame(*index*)

Remove a frame from the animation.

That method remove the frame at the specified index and return it if it exists.

If the index is out of bound an exception is raised. If the index is not an int an exception is raised.

Parameters
index (*int*) – The index of the frame to remove.

Return type
str

Raise
IndexError, PglInvalidTypeException

Example:

```
item.animation.remove_frame( item.animation.search_frame(
    Sprite.ALIEN_MONSTER)
)
```

reset()

Reset the Animation to the first frame.

Example:

```
item.animation.reset()
```

search_frame(*frame*)

Search a frame in the animation.

That method is returning the index of the first occurrence of “frame”.

Raise an exception if frame is not a string.

Parameters
frame (*str*) – The frame to find.

Return type
int

Raise
PglInvalidTypeException

Example:

```
item.animation.remove_frame(
    item.animation.search_frame(Sprite.ALIEN_MONSTER)
)
```

serialize()

Serialize the Animation object.

The *refresh_screen* callback function is not serialized. Neither is the parent.

Returns
A dictionary containing the Animation object’s data.

Return type
dict

start()

Set the animation state to State.RUNNING.

If the animation state is not State.RUNNING, animation's next_frame() function return the last frame returned.

Example:

```
item.animation.start()
```

stop()

Set the animation state to STOPPED.

Example:

```
item.animation.stop()
```

Font

class pygamelib.gfx.core.**Font**(*font_name: str = None*, *search_directories: list = None*)

Bases: `object`

New in version 1.3.0.

The Font class allow to load and manipulate a pygamelib "font". A font consist of a sprite collection and a configuration file.

If you want to create your own font, please have a look at the font creation tutorial.

In general the Font class is not used directly but passed to a `Text` object. The text is then rendered using the font.

For performance consideration, it is advised to load the font once and to reuse the object in multiple text objects.

Glyphs are cached (particularly if you change the colors) so it is always beneficial to reuse a font object.

Example:

```
myfont = Font("8bits")
# If you print() mytext, it will use the terminal font and print in cyan.
# But if you Sreen.place() it, it will render using the 8bits sprite font.
mytext = Text("Here's a cool text", fg_color = Color(0,255,255), font=myfont)
```

__init__(*font_name: str = None*, *search_directories: list = None*) → None

Parameters

- **font_name** (*str*) – The name of the font to load upon object construction.
- **search_directories** (*list*) – A list of directories to search for the font. The items of the list are strings representing a relative or absolute path.

Important: The search directories **must** contain a "fonts" directory, that itself contains the font at the correct format.

Note: Version 1.3.0 comes with a pygamelib specific font called 8bits. It also comes with a handfull of fonts imported from the figlet fonts. Please go to http://www.figlet.org/ for more information.

The conversion script will be made available in the Pygamelib Github organization (https://github.com/pygamelib).

Example:

```
myfont = Font("8bits")
```

Methods

`__init__`([font_name, search_directories])	**param font_name** The name of the font to load upon object construction.
`glyph`([glyph_name, fg_color, bg_color])	This method take a glyph name in parameter and returns its representation as a `Sprite`.
`load`([font_name])	Load a font by name.

Attributes

`colorable`	Returns the "colorability" of the font as specified in the font config file.
`glyphs_map`	Returns the glyph map of the font as specified in the font config file.
`height`	Returns the height of the font as specified in the font config file.
`horizontal_spacing`	Returns the horizontal spacing recommended by the font (as specified in the font config file).
`monospace`	Returns if the font is monospace as specified in the font config file.
`name`	Return the name of the font.
`scalable`	Returns the scalability of the font as specified in the font config file.
`vertical_spacing`	Returns the vertical spacing recommended by the font (as specified in the font config file).

property colorable: bool

Returns the "colorability" of the font as specified in the font config file.

Return type
bool

glyph(*glyph_name: str = None*, *fg_color: Color = None*, *bg_color: Color = None*) → *Sprite*

This method take a glyph name in parameter and returns its representation as a `Sprite`.

The glyph name is usually the name of a character (like "a") but it is not mandatory and can be anything. The default glyph (returned when no glyph matches the requested glyph) is called "default" for example.

Parameters
glyph_name (*str*) – The glyph name

Returns
A glyphe as a *Sprite*

Return type
Sprite

Example:

```
myfont = Font("8bits")
row = 5
column = 10
for letter in "this is a text":
    glyph = myfont.glyph(letter)
    screen.place(glyph, row, column)
    column += glyph.width + myfont.horizontal_spacing()

# Please note that in real life you would just do this
mytext = Text("this is a text", font=myfont)
screen.place(mytext, row, column)
```

property glyphs_map: dict

Returns the glyph map of the font as specified in the font config file.

Return type
dict

property height: int

Returns the height of the font as specified in the font config file.

Return type
int

Example:

```
screen.place(text, last_row + myfont.height, first_text_column)
```

property horizontal_spacing: int

Returns the horizontal spacing recommended by the font (as specified in the font config file).

As a user of the font class using the Font class to change the look of some text, you will rarely use that value directly (it is directly used by Text.render_to_buffer()).

If your goal is to use the Font class to do glyph rendering as you see fit, use the horizontal spacing value to place each glyph relatively to the one on its left or right.

Return type
int

load(*font_name: str = None*) → None

Load a font by name. Once the font is loaded glyphs can be accessed through the *glyph()* method.

This method is automatically called is the Font constructor is called with a font name.

Parameters
font_name (*str*) – The name of the font to load upon object construction.

Example:

```
# The 2 following examples do exactly the same thing.
# Example 1: instantiate and load
myfont = Font()
myfont.load("8bits")
# Example 2: load from instantiation
myfont2 = Font("8bits")
# At that point myfont and myfont2 are exactly the same (and there is no
# good justification to instantiate or load the font twice).
```

property monospace: bool

Returns if the font is monospace as specified in the font config file.

Return type
bool

property name: str

Return the name of the font. The name is the string that was used to load the font.

Example:

```
myfont = Font("8bits")
if myfont.name() != "8bits":
    print("Something very wrong just occurred!")
```

property scalable: bool

Returns the scalability of the font as specified in the font config file.

Return type
bool

property vertical_spacing: int

Returns the vertical spacing recommended by the font (as specified in the font config file).

Return type
int

Example:

```
screen.place(
    text,
    last_row + myfont.height() + myfont.vertical_spacing(),
    first_text_column
)
```

SpriteCollection

class pygamelib.gfx.core.**SpriteCollection**(*data=None*)

Bases: `UserDict`

SpriteCollection is a dictionnary class that derives collections.UserDict.

Its main goal is to provide an easy to use object to load and save sprite files. On top of traditional dict method, it provides the following capabilities:

- loading and writing from and to JSON files,
- data serialization,

- shortcut to add sprites to the dictionnary.

A SpriteCollection is an unordered indexed list of Sprites (i.e a dictionnary).

Sprites are indexed by their names in that collection.

Example:

```
# Load a sprite file
sprites_village1 = SpriteCollection.load_json_file('gfx/village1.spr')
# display the Sprites with their name
for sprite_name in sprites_village1:
    print(f'{sprite_name}:\n{sprites_village1[sprite_name]}')
# Add an empty sprite with name 'house_placeholder'
sprites_village1.add( Sprite(name='house_placeholder') )
# This is absolutely equivalent to:
sprites_village1['house_placeholder'] = Sprite(name='house_placeholder')
# And now rewrite the sprite file with the new placeholder house
sprites_village1.to_json_file('gfx/village1.spr')
```

__init__(*data=None*)

Methods

__init__([data])	
add(sprite)	Add a Sprite to the collection.
clear()	
copy()	
fromkeys(iterable[, value])	
get(k[,d])	
items()	
keys()	
load(data)	Load serialized data and return a new SpriteCollection object.
load_json_file(filename)	Load a JSON sprite file into a new SpriteCollection object.
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
rename(old_key, new_key)	Rename a key in the collection.
serialize()	Return a serialized version of the SpriteCollection.
setdefault(k[,d])	
to_json_file(filename)	Export the SpriteCollection object in JSON and writes it on the disk.
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

add(*sprite*)

Add a Sprite to the collection. This method is simply a shortcut to the usual dictionnary affectation. The collection requires the name of the Sprite to be the key. That method does that automatically.

Parameters

sprite (*Sprite*) – A Sprite object to add to the collection.

> **Warning:** As SpriteCollection index Sprites by their name if you change the Sprite's name *after* adding it to the collection you will need to manually update the keys.

Example:

```
sprites_village1 = SpriteCollection.load_json_file('gfx/village1.spr')
new_village = SpriteCollection()
```

(continues on next page)

(continued from previous page)

```
new_village.add( copy.deepcopy( sprites_village1.get('bakery') ) )
print( new_village['bakery'] )
```

clear() → None. Remove all items from D.

copy()

classmethod fromkeys(*iterable*, *value=None*)

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

classmethod load(*data*)

Load serialized data and return a new SpriteCollection object.

Parameters
data (*str*) – Serialized data that need to be expanded into objects.

Returns
A new SpriteCollection object.

Return type
SpriteCollection

Example:

```
sprites_village1 = SpriteCollection.load(
    sprites_village_template.serialize()
)
```

static load_json_file(*filename*)

Load a JSON sprite file into a new SpriteCollection object.

Parameters
filename (*str*) – The complete path (relative or absolute) to the sprite file.

Returns
A new SpriteCollection object.

Return type
SpriteCollection

Example:

```
sprites_village1 = SpriteCollection.load_json_file('gfx/village1.spr')
```

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

rename(*old_key*, *new_key*)

Rename a key in the collection.

This methods also takes care of renaming the Sprite associated with the old key name.

Parameters

- **old_key** (*str*) – The key to rename
- **new_key** (*str*) – The new key name

Example:

```
my_collection.rename('panda', 'panda walk 01')
```

serialize()

Return a serialized version of the SpriteCollection. The serialized data can be pass to the JSON module to export.

Returns
The SpriteCollection object serialized as a dictionnary.

Return type
dict

Example:

```
data = sprites_village1.serialize()
```

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

to_json_file(*filename*)

Export the SpriteCollection object in JSON and writes it on the disk.

Parameters
filename (*str*) – The complete path (relative or absolute) to the sprite file to write.

Example:

```
sprites_village1.to_json_file('gfx/village1.spr')
```

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

Sprite

class pygamelib.gfx.core.**Sprite**(*sprixels=None*, *default_sprixel=None*, *parent=None*, *size=[2, 2]*, *name=None*)

Bases: *PglBaseObject*

The Sprite object represent a 2D "image" that can be used to represent any complex item. Obviously, a sprite in the pygamelib is not really an image, it is a series of glyphs (or characters) with colors (foreground and background) information.

A Sprite object is a 2D array of *Sprixel*.

If you use the climage python module, you can load the generated result into a Sprite through Sprite.load_from_ansi_file().

Parameters

- **sprixels** (*list*) – A 2D array of *Sprixel*.

- **default_sprixel** (*Sprixel*) – A default Sprixel to complete lines that are not long enough. By default, it's an empty Sprixel.
- **parent** (*BoardComplexItem* (suggested)) – The parent object of this Sprite. If it's left to None, the *BoardComplexItem* constructor takes ownership of the sprite.
- **size** (*list*) – A 2 elements list that represent the width and height ([width, height]) of the Sprite. It is only needed if you create an empty Sprite. If you load from a file or provide an array of sprixels it's obviously calculated automatically. Default value: [2, 2].
- **name** (*str*) – The name of sprite. If none is given, an UUID will be automatically generated.

Example:

```
void = Sprixel()
# This represent a panda
panda_sprite = Sprite(
    sprixels=[
        [void, void, void, void, void, void, void, void],
        [
            Sprixel.black_rect(),
            Sprixel.black_rect(),
            void,
            void,
            void,
            void,
            Sprixel.black_rect(),
            Sprixel.black_rect(),
        ],
        [
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
        ],
        [
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.black_rect(),
            Sprixel.black_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.black_rect(),
            Sprixel.black_rect(),
        ],
        [
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.red_rect(),
```

(continues on next page)

(continued from previous page)

```
            Sprixel.red_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
        ],
        [
            void,
            void,
            Sprixel.black_rect(),
            Sprixel.black_rect(),
            Sprixel.black_rect(),
            Sprixel.black_rect(),
            void,
            void,
        ],
        [
            void,
            void,
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.white_rect(),
            Sprixel.black_rect(),
            Sprixel.black_rect(),
        ],
        [
            void,
            void,
            Sprixel.black_rect(),
            Sprixel.black_rect(),
            void,
            void,
            void,
            void,
        ],
    ],
)
```

`__init__`(*sprixels=None*, *default_sprixel=None*, *parent=None*, *size=[2, 2]*, *name=None*)

Like the object class, this class constructor takes no parameter.

Methods

`__init__`([sprixels, default_sprixel, ...])	Like the object class, this class constructor takes no parameter.
`attach`(observer)	Attach an observer to this instance.
`calculate_size`()	Calculate the size of the sprite and update the size variable.
`copy`()	Returns a (deep) copy of the sprite.
`detach`(observer)	Detach an observer from this instance.
`empty`()	Empty the sprite and fill it with default sprixels.
`flip_horizontally`()	Flip the sprite horizontally.
`flip_vertically`()	Flip the sprite vertically (i.e upside/down).
`from_text`(text_object)	Create a Sprite from a `Text` object.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Create a new Sprite object based on serialized data.
`load_from_ansi_file`(filename[, default_sprixel])	Load an ANSI encoded file into a Sprite object.
`modulate`(color[, ratio])	Modulate the sprite colors with the color in parameters.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`render_to_buffer`(buffer, row, column, ...)	Render the sprite from the display buffer to the frame buffer.
`scale`([ratio])	Scale a sprite up and down using the nearest neighbor algorithm.
`serialize`()	Serialize a Sprite into a dictionary.
`set_sprixel`(row, column, value)	Set a specific sprixel in the sprite to the given value.
`set_transparency`(state)	This method enable transparent background to all the sprite's sprixels.
`sprixel`([row, column])	Return a sprixel at a specific position within the sprite.
`store_screen_position`(row, column)	Store the screen position of the object.
`tint`(color[, ratio])	Tint a copy of the sprite with the color.

Attributes

`height`	Property that returns the height of the Sprite.
`screen_column`	A property to get/set the screen column.
`screen_row`	A property to get/set the screen row.
`width`	Property that returns the width of the Sprite.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (`PglBaseObject`) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

calculate_size()

Calculate the size of the sprite and update the size variable.

The size is immediately returned.

It is done separately for concerns about performances of doing that everytime the size is requested.

Return type
list

Example:

```
spr_size = spr.calculate_size()
if spr_size != spr.size:
    raise PglException(
                'perturbation_in_the_Force',
                'Something is very wrong with the sprite!'
            )
```

copy()

Returns a (deep) copy of the sprite.

New in version 1.3.0.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

empty()

Empty the sprite and fill it with default sprixels.

Example:

```
player_sprite.empty()
```

flip_horizontally()

Flip the sprite horizontally.

This method performs a symmetry versus the vertical axis.

At the moment, glyph are not inverted. Only the position of the sprixels.

The flipped sprite is returned (original sprite is not modified).

Return type
Sprite

Example:

```
reflection_sprite = player_sprite.flip_horizontally()
```

flip_vertically()

Flip the sprite vertically (i.e upside/down).

At the moment, glyph are not inverted. Only the position of the sprixels. There is one exception however, as climage uses the ‘’ utf8 glyph as a marker, that specific glyph is inverted to ‘’ and vice versa.

The flipped sprite is returned (original sprite is not modified).

Return type
Sprite

Example:

```
reflection_sprite = player_sprite.flip_vertically()
```

classmethod **from_text**(*text_object*)

Create a Sprite from a *Text* object.

Parameters
text_object (*Text*) – A text object to transform into Sprite.

Example:

```
# The Text object allow for easy manipulation of text
village_name = base.Text('Khukdale',fg_red, bg_green)
# It can be converted into a Sprite to be displayed on the Board
village_sign = board_items.Tile(sprite=Sprite.from_text(village_name))
# And can be used as formatted text
notifications.push( f'You enter the dreaded village of {village_name}' )
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event’s value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a “FQDN style” string. This can be None.

- **value** (*Any*) – The new value of the attribute. This can be None.

property height

Property that returns the height of the Sprite.

New in version 1.3.0.

Contrary to Sprite.size[1], this property *always* calls Sprite.calculate_size() before returning the height.

classmethod load(*data*)

Create a new Sprite object based on serialized data.

New in version 1.3.0.

Parameters
data (*dict*) – Data loaded from a JSON sprite file (deserialized).

Return type
Sprite

Example:

```
new_sprite = Sprite.load(json_parsed_data)
```

classmethod load_from_ansi_file(*filename*, *default_sprixel=None*)

Load an ANSI encoded file into a Sprite object.

This class method can load a file produced by the climage python module and load it into a Sprite class. Each character is properly decoded into a *Sprixel* with model, background and foreground colors.

A Sprite is rectangular (at least for the moment), so in case the file is not shaped as a rectangle, this method automatically fills the void with a default sprixel (to make sure all lines in the sprite have the same length). By default, it fills the table with None "values" but you can specify a default sprixel.

The reasons the default sprixel is set to None is because None values in a sprite are not translated into a component in *BoardComplexItem* (i.e no sub item is generated).

Parameters

- **filename** (*str*) – The path to a file to load.
- **default_sprixel** (None | *Sprixel*) – The default Sprixel to fill a non rectangular shaped sprite.

Example:

```
player_sprite = gfx_core.Sprite.load_from_ansi_file('gfx/models/player.ans')
```

modulate(*color: Color*, *ratio: float = 0.5*)

Modulate the sprite colors with the color in parameters.

New in version 1.3.0.

This method tint all the sprixels of the sprite with the color at the specified ratio. **The original sprite IS modified**.

If you want to keep the original sprite intact consider using *tint()*.

Parameters

- **color** (*Color*) – The modulation color.
- **ratio** (*float*) – The modulation ratio between 0.0 and 1.0 (default: 0.5)

Returns
None

When this method is called, the observers are notified of the change with the pygamelib.core.Sprite.color:modulated event. No arguments are passed along this event.

Example:

```
player_sprites = core.SpriteCollection.load_json_file("gfx/player.spr")
# After that, the sprite is quite not "normal" anymore...
player_sprites["normal"].modulate(core.Color(0, 255, 0), 0.3)
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render the sprite from the display buffer to the frame buffer.

New in version 1.3.0.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

scale(*ratio=1.0*)

Scale a sprite up and down using the nearest neighbor algorithm.

New in version 1.3.0.

Parameters
ratio (*float*) – The scaling ration.

Returns
An upscaled/downscaled sprite.

Return type
Sprite

Note: The sprites generated with pgl-converter.py don't scale well yet if the –unicode flag is active

Example:

```
bigger_sprite = original_sprite.scale(2)
```

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a Sprite into a dictionary.

New in version 1.3.0.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( sprite.serialize() )
```

set_sprixel(*row*, *column*, *value*)

Set a specific sprixel in the sprite to the given value.

Parameters

- **row** (*int*) – The row of the sprite (WARNING: internal sprite coordinates)
- **column** (*int*) – The column of the sprite (same warning)
- **value** (*Sprixel*) – The sprixel to set at [row, column]

When a sprixel is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprite.sprixel:changed event. A structure is passed as the *value* parameter. This structure has 3 members: row, column and sprixel.

Example:

```
my_sprite.set_sprixel(1, 2, Sprixel("#",fg_color=green))
```

set_transparency(*state*)

This method enable transparent background to all the sprite's sprixels.

New in version 1.3.0.

Parameters
state – a boolean to enable or disable background transparency

When the transparency is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprite.transparency:changed event. The new transparency state is passed as the *value* parameter.

Example:

```
player_sprite.set_transparency(True)
```

Warning: This set background transparency on all sprixels, make sure you are not using background colors as part of your sprite before doing that. It can also be used as a game/rendering mechanic. Just make sure you know what you do. As a reminder, by default, sprixels with no background have transparent background enable.

sprixel(*row=0*, *column=None*)

Return a sprixel at a specific position within the sprite.

If the column is set to None, the whole row is returned.

Parameters

- **row** (*int*) – The row to access within the sprite.
- **column** (*int*) – The column to access within the sprite.

Returns
Sprixel

Example:

```
# Return the entire line at row index 2
scanline = house_sprite.sprixel(2)
# Return the specific sprixel at sprite internal coordinate 2,3
house_sprixel = house_sprite.sprixel(2, 3)
```

Warning: For performance consideration sprixel() does not check the size of its matrix. This method is called many times during rendering and 2 calls to len() in a row are adding up pretty quickly. It checks the boundary of the sprite using the cached size. Make sure it is up to date!

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

tint(*color: Color*, *ratio: float = 0.5*)

Tint a copy of the sprite with the color.

New in version 1.3.0.

This method creates a copy of the sprite and tint all its sprixels with the color at the specified ratio. It then returns the new sprite. **The original sprite is NOT modified**.

Parameters

- **color** (*Color*) – The tint color.
- **ratio** (*float*) – The tint ration between 0.0 and 1.0 (default: 0.5)

Returns

Sprite

Example:

```
player_sprites = core.SpriteCollection.load_json_file("gfx/player.spr")
player_sprites["sick"] = player_sprites["normal"].tint(
                            core.Color(0, 255, 0), 0.3
                        )
```

property width

Property that returns the width of the Sprite.

New in version 1.3.0.

Contrary to Sprite.size[0], this property *always* calls Sprite.calculate_size() before returning the width.

Sprixel

class pygamelib.gfx.core.**Sprixel**(*model=''*, *bg_color=None*, *fg_color=None*, *is_bg_transparent=None*)

Bases: *PglBaseObject*

A sprixel is the representation of 1 cell of the sprite or one cell on the Board. It is not really a pixel but it is the closest notion we'll have. A Sprixel has a background color, a foreground color and a model. All regular BoardItems can now use a sprixel instead of a model (but simple model is still supported of course).

In the terminal, a sprixel is represented by a single character.

If the background color and the is_bg_transparent are None, the sprixel will be automatically configured with transparent background. In that case, as we cannot really achieve transparency in the console, the sprixel will take the background color of whatever it is overlapping.

Important: BREAKING CHANGE: in version 1.3.0 background and foreground colors use the new *Color* object. Therefor, Sprixel does not accept ANSI sequences anymore for the bg_color and fg_color parameters.

Example:

```
player = Player(sprixel=Sprixel(
                                '#',
                                Color(128,56,32),
                                Color(255,255,0),
                                ))
```

__init__(*model=''*, *bg_color=None*, *fg_color=None*, *is_bg_transparent=None*)

Parameters

- **model** (*str*) – The model, it can be any string. Preferrably a single character.
- **bg_color** (*Color*) – A Color object to configure the background color.
- **fg_color** (*Color*) – A Color object to configure the foreground color.
- **is_bg_transparent** (*bool*) – Set the background of the Sprixel to be transparent. It tells the engine to replace the background of the Sprixel by the background color of the overlapped sprixel.

Methods

__init__([model, bg_color, fg_color, ...])	**param model** The model, it can be any string. Preferrably a single character.
attach(observer)	Attach an observer to this instance.
black_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_RECT.
black_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_SQUARE.
blue_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_RECT.
blue_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_SQUARE.
copy()	Returns a (deep) copy of the sprixel.
cyan_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_RECT.
cyan_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_SQUARE.
detach(observer)	Detach an observer from this instance.
from_ansi(string[, model])	Takes an ANSI string, parse it and return a Sprixel.
green_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_RECT.
green_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_SQUARE.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Create a new Sprixel object based on serialized data.
magenta_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_RECT.
magenta_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_SQUARE.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
red_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_RECT.
red_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_SQUARE.
render_to_buffer(buffer, row, column, ...)	Render the sprixel from the display buffer to the frame buffer.
serialize()	Serialize a Sprixel into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
white_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_RECT.
white_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_SQUARE.
yellow_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_RECT.
yellow_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_SQUARE.

Attributes

bg_color	A property to get/set the background color of the Sprixel.
fg_color	A property to get/set the foreground color of the Sprixel.
length	Return the true length of the model.
model	A property to get/set the model of the Sprixel.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property bg_color

A property to get/set the background color of the Sprixel.

Parameters
value (*Color*) – The new color

When the bg_color is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprixel.bg_color:changed event. The new bg_color is passed as the *value* parameter.

Example:

```
# Access the sprixel's color
sprix.bg_color
# Set the sprixel's background color to some blue
sprix.bg_color = Color(0,128,255)
```

classmethod black_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_RECT. The difference is that BLACK_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.black_rect()
```

classmethod black_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_SQUARE. The difference is that BLACK_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.black_square()
```

classmethod blue_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_RECT. The difference is that BLUE_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.blue_rect()
```

classmethod blue_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_SQUARE. The difference is that BLUE_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.blue_square()
```

copy()

Returns a (deep) copy of the sprixel.

New in version 1.3.0.

classmethod cyan_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_RECT. The difference is that CYAN_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.cyan_rect()
```

classmethod cyan_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_SQUARE. The difference is that CYAN_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.cyan_square()
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

property fg_color

A property to get/set the foreground color of the Sprixel.

Parameters
value (*Color*) – The new color

When the fg_color is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprixel.fg_color:changed event. The new fg_color is passed as the *value* parameter.

Example:

```
# Access the sprixel's color
sprix.fg_color
# Set the sprixel's foreground color to some green
sprix.fg_color = Color(0,255,128)
```

static from_ansi(*string*, *model=''*)

Takes an ANSI string, parse it and return a Sprixel.

Parameters

- **string** (*str*) – The ANSI string to parse.
- **model** (*str*) – The character used to represent the sprixel in the ANSI sequence. Default is “”

Example:

```
new_sprixel = Sprixel.from_ansi(
    "\x1b[48;2;139;22;19m\x1b[38;2;160;26;23m\x1b[0m"
)
```

Warning: This has mainly be tested with ANSI string generated by climage. If you find any issue, please report it

classmethod green_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_RECT. The difference is that GREEN_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.green_rect()
```

classmethod green_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_SQUARE. The difference is that GREEN_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.green_square()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property length

Return the true length of the model.

New in version 1.3.0.

With UTF8 and emojis the length of a string as returned by python's `len()` function is often very wrong. For example, the len("x1b[48;2;139;22;19mx1b[38;2;160;26;23mx1b[0m") returns 39 when it should return 1.

This method returns the actual printing/display size of the sprixel's model.

Note: This is a read only value. It is automatically updated when the model is changed.

Example:

```
if sprix.length > 2:
    print(
        f"Warning: that sprixel {sprix} will break the rest of the "
        "board's alignement"
        )
```

classmethod load(*data*)

Create a new Sprixel object based on serialized data.

New in version 1.3.0.

Parameters
data (*dict*) – Data loaded from JSON data (deserialized).

Return type
Sprixel

Example:

```
new_sprite = Sprixel.load(json_parsed_data['default_sprixel'])
```

classmethod magenta_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_RECT. The difference is that MAGENTA_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.magenta_rect()
```

classmethod magenta_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_SQUARE. The difference is that MAGENTA_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.magenta_square()
```

property model

A property to get/set the model of the Sprixel.

Parameters

value (*str*) – The new model

When the model is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprixel.model:changed event. The new model is passed as the *value* parameter.

Example:

```
# Get the sprixel's model
sprix.model
# Set the sprixel's model to "@"
sprix.model = "@"
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

classmethod red_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_RECT. The difference is that RED_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.red_rect()
```

classmethod red_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_SQUARE. The difference is that RED_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.red_square()
```

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render the sprixel from the display buffer to the frame buffer.

New in version 1.3.0.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a Sprixel into a dictionary.

New in version 1.3.0.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( sprixel.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

classmethod **white_rect**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_RECT. The difference is that WHITE_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.white_rect()
```

classmethod **white_square**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_SQUARE. The difference is that WHITE_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.white_square()
```

classmethod **yellow_rect**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_RECT. The difference is that YELLOW_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Note: Yellow is often rendered as brown.

Example:

```
sprixel = Sprixel.yellow_rect()
```

classmethod **yellow_square**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_SQUARE. The difference is that YELLOW_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Note: Yellow is often rendered as brown.

Example:

```
sprixel = Sprixel.yellow_square()
```

Color

class pygamelib.gfx.core.**Color**(*r=0*, *g=0*, *b=0*)

Bases: *PglBaseObject*

New in version 1.3.0.

A color represented by red, green and blue (RGB) components. Values are integer between 0 and 255 (both included).

Parameters

- **r** (*int*) – The red component of the color.
- **g** (*int*) – The green component of the color.
- **b** (*int*) – The blue component of the color.

Example:

```
# color is blue
color = Color(0, 0, 255)
# and now color is pink
color.r = 255
```

__init__(*r=0*, *g=0*, *b=0*)

Like the object class, this class constructor takes no parameter.

Methods

__init__([r, g, b])	Like the object class, this class constructor takes no parameter.
attach(observer)	Attach an observer to this instance.
blend(other_color[, fraction])	Blend the color with another one.
copy()	Returns a (deep) copy of this color.
detach(observer)	Detach an observer from this instance.
from_ansi(string)	Create and return a Color object based on an ANSI color string.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Create a new Color object based on serialized data.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
random()	Create and return a new random color.
randomize()	Set a random value for each of the components of an existing color.
serialize()	Serialize a Color into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

b	The b property controls the intensity of the blue color.
g	The g property controls the intensity of the green color.
r	The r property controls the intensity of the red color.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property b

The b property controls the intensity of the blue color. You can set it to an integer between 0 and 255 (both included).

When this property is set, the observers are notified with the pygamelib.gfx.core.Color.b:changed event. The value of the event is the new value of the property.

Example:

```
color = Color(128, 128, 0)
print(f"Value for b is {color.b}")
color.b = 255
print(f"New value for b is {color.b}")
```

blend(*other_color*, *fraction=0.5*)

Blend the color with another one. Fraction controls the amount of other_color that is included (0 means no inclusion at all).

Parameters

- **other_color** (*Color*) – The color to blend with.
- **fraction** (*float*) – The blending modulation factor between 0 and 1.

Returns

A new Color object that contains the blended color.

Return type
`Color`

Example:

```
a = Color(200, 200, 200)
b = Color(25, 25, 25)
# c is going to be Color(112, 112, 112)
c = a.blend(b, 0.5)
```

copy()

Returns a (deep) copy of this color.

Example:

```
red = Color(255, 0, 0)
red2 = red.copy()
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

classmethod from_ansi(*string*)

Create and return a Color object based on an ANSI color string.

Important: The string must be RGB, i.e '[38;2;RED;GREEN;BLUEm' or '[48;2;RED;GREEN;BLUEm' for foreground and background colors. This method will return None if the color string is not RGB. It is also important to understand that Color is independent from the foreground of background, it is just a color. Therefor '[38;2;89;32;93m' and '[48;2;89;32;93m' will both be parsed into Color(89, 32, 93).

Parameters
string (*str*) – The ANSI color string to convert.

Example:

```
color = Color.from_ansi()
```

property g

The g property controls the intensity of the green color. You can set it to an integer between 0 and 255 (both included).

When this property is set, the observers are notified with the pygamelib.gfx.core.Color.g:changed event. The value of the event is the new value of the property.

Example:

```
color = Color(128, 128, 0)
print(f"Value for g is {color.g}")
color.g = 255
print(f"New value for g is {color.g}")
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Create a new Color object based on serialized data.

If data is None, None is returned.

If a color component is missing from data, it is set to 0 (see examples).

Raises an exception if the color components are not integer.

Parameters

data (*dict*) – Data loaded from JSON data (deserialized).

Returns

Either a Color object or None if data where empty.

Return type

Color | NoneType

Raise

PglInvalidTypeException

Example:

```
# Loading from parsed JSON data
new_color = Color.load(json_parsed_data['default_sprixel']['fg_color'])

# Loading from incomplete data
color = Color.load({'red':25,'green':35})
# Result in the following Color object:
# Color(25, 35, 0)
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property r

The r property controls the intensity of the red color. You can set it to an integer between 0 and 255 (both included).

When this property is set, the observers are notified with the pygamelib.gfx.core.Color.r:changed event. The value of the event is the new value of the property.

Example:

```
color = Color(128, 128, 0)
print(f"Value for r is {color.r}")
color.r = 255
print(f"New value for r is {color.r}")
```

classmethod random()

Create and return a new random color.

Return type
Color

Example:

```
my_color = Color.random()
```

randomize()

Set a random value for each of the components of an existing color.

When this method is called, the observers are notified with the pygamelib.gfx.core.Color.randomized event. The value of the event is the new color.

Returns
None

Return type
NoneType

Example:

```
color = Color()
color.randomize()
```

property screen_column: int

A property to get/set the screen column.

Parameters
`value` (*int*) – the screen column

Return type
int

`property screen_row: int`

A property to get/set the screen row.

Parameters
`value` (*int*) – the screen row

Return type
int

`serialize()`

Serialize a Color into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( color.serialize() )
```

`store_screen_position`(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- `row` (*int*) – The row (or y) coordinate.
- `column` (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

3.7.2 ui

Warning: The UI module is in alpha version. Some things might change over time.

The ui module contains the classes to easily build full screen Terminal User Interface (TUI) for your games (or applications).

Important: It works exclusively with the screen buffer system (place, delete, render, update, etc.). It doesn't work with Screen functions tagged "direct display" like display_at().

Box

class pygamelib.gfx.ui.**Box**(*width: int*, *height: int*, *title: str = ''*, *config: UiConfig = None*, *fill: bool = False*, *filling_sprixel: Sprixel = None*, *title_alignment: Alignment = Alignment.CENTER*)

Bases: `object`

A simple object to draw a box on screen.

The Box object's looks and feel is highly configurable through the *UiConfig* object.

__init__(*width: int*, *height: int*, *title: str = ''*, *config: UiConfig = None*, *fill: bool = False*, *filling_sprixel: Sprixel = None*, *title_alignment: Alignment = Alignment.CENTER*)

The box constructor takes the following parameters.

Parameters

- **width** (*int*) – The width of the box.
- **height** (*int*) – The height of the box.
- **title** (str | `Text`) – The title of the box (encased in the top border).
- **config** (*UiConfig*) – The configuration object.
- **fill** (*bool*) – A tag to tell the box object to fill its inside (or not).
- **filling_sprixel** (`Sprixel`) – If fill is True, the filling Sprixel is used to fill the inside of the box.
- **title_alignment** (*int*) – The alignment of the title in the top bar. It is a constant from the constant module and can be ALIGN_LEFT, ALIGN_RIGHT and ALIGN_CENTER. THIS FEATURE IS NOT YET IMPLEMENTED.

Todo: Implement the title alignment.

Example:

```
config = UiConfig(bg_color=None)
box = Box(30, 10, 'This is a box')
screen.place(box, 20, 20)
screen.update()
```

Methods

__init__(width, height[, title, config, ...])	The box constructor takes the following parameters.
render_to_buffer(buffer, row, column, ...)	Render the box from the display buffer to the frame buffer.

Attributes

config	Get and set the config object (*UiConfig*).
height	Get and set the height of the box, only accept int.
title	Get and set the title, only accepts str or Text.
width	Get and set the width of the box, only accept int.

property config: *UiConfig*

Get and set the config object (*UiConfig*).

property height: int

Get and set the height of the box, only accept int.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the box from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property title

Get and set the title, only accepts str or Text.

property width: int

Get and set the width of the box, only accept int.

BoxLayout

class pygamelib.gfx.ui.**BoxLayout**(*orientation:* Orientation | *None = None*, *size_constraint:* SizeConstraint | *None = None*, *parent:* Widget | *None = None*)

Bases: *Layout*

The box layout lines up child widgets horizontally or vertically. The orientation of the layout is controlled using the *Orientation* constants.

__init__(*orientation:* Orientation | *None = None*, *size_constraint:* SizeConstraint | *None = None*, *parent:* Widget | *None = None*) → None

Parameters

- **orientation** (*Orientation*) – The orientation of the layout.
- **size_constraint** (*SizeConstraint*) – The size constraint policy for managed widgets.
- **parent** (*Widget*) – The parent object, ie: the one in which the GridLayout reside.

Example:

```
parent_widget = Widget(45, 30, config=config)
# Add a GridLayout to a widget
parent_widget.layout = BoxLayout(orientation=Orientation.VERTICAL)
parent_widget.layout.add_widget(LineInput())
```

Methods

__init__([orientation, size_constraint, parent])	**param orientation** The orientation of the layout.
add_widget(w)	Add a widget to the BoxLayout.
attach(observer)	Attach an observer to this instance.
count()	Returns the amount (the count) of widgets in the BoxLayout.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
store_screen_position(row, column)	Store the screen position of the object.
widgets()	Returns the list of widgets that are managed by the GridLayout as a set.

Attributes

height	Get the layout's height (including spacing).
orientation	Get and set the layout's orientation.
parent	This property get/set the parent of the Layout (if any).
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size_constraint	Get and set the layout's size constraint policy.
spacing	This property get/set the inter-widgets spacing of the Layout.
width	Get the layout's width (including spacing).

add_widget(*w:* Widget) → bool

Add a widget to the BoxLayout. If the widget is correctly added to the layout this method returns True, otherwise it returns False.

Parameters

widget (class:*Widget*) – The widget to add to the layout.

Example:

```
parent_widget = Widget(45, 30, config=config)
# Add a BoxLayout to a widget
parent_widget.layout = BoxLayout()
```

(continues on next page)

(continued from previous page)

```
# Then add children widgets to the parent's layout
# Step 1: create a widget (here just a generic widget)
child_widget1 = Widget(config=UiConfig.instance())
# Step 2: add it to the layout.
parent_widget.layout.add_widget(child_widget1)
# That's it!
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

count() → int

Returns the amount (the count) of widgets in the BoxLayout.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property height: int

Get the layout's height (including spacing).

Returns an int.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property orientation: *Orientation*

Get and set the layout's orientation.

The orientation of the BoxLayout can be changed dynamically and will take effect immediately.

It has to be a *Orientation*.

property parent: *Widget* | None

This property get/set the parent of the Layout (if any).

render_to_buffer(*buffer: numpy.array*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

property size_constraint: *SizeConstraint*

Get and set the layout's size constraint policy. It has to be a *SizeConstraint*.

property spacing: int

This property get/set the inter-widgets spacing of the Layout.

When the spacing is changed the pygamelib.gfx.ui.Layout.spacing:changed event is sent to the observers.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

widgets() → Set[*Widget*]

Returns the list of widgets that are managed by the GridLayout as a set. This set is not guaranteed to be ordered!

property width: int

Get the layout's width (including spacing).

Returns an int.

ColorPickerDialog

class pygamelib.gfx.ui.**ColorPickerDialog**(*title: str = None*, *config:* UiConfig *= None*)

Bases: *Dialog*

The ColorPickerDialog is a dialog wrapper around the *ColorPicker* widget.

It serves the same purpose: present a way to easily select a custom color to the user.

It does it as an immediately usable dialog.

The show() method returns the *Color* selected by the user. If the user pressed the ESC key, it returns None.

Key mapping:

- ESC: Exit from the show() method and return None.
- ENTER: Exit from the show() method. Returns the currently selected color.
- UP / DOWN: Increase/decrease the currently selected channel by 1.
- PAGE_UP / PAGE_DOWN: Increase/decrease the currently selected channel by 10.
- LEFT / RIGHT: Navigate between color channels.

Like all dialogs, it is automatically destroyed on exit of the *show()* method. It is also deleted from the screen buffer.

`__init__`(*title: str = None*, *config: UiConfig = None*) → None

The constructor only take the configuration as parameter.

Parameters

config (*UiConfig*) – The configuration object.

Example:

```
color_dialog = ColorPickerDialog(conf)
color_dialog.set_color( core.Color(128, 128, 128) )
screen.place(color_dialog, 10, 10)
new_color = color_dialog.show()
```

Methods

__init__([title, config])	The constructor only take the configuration as parameter.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
set_color(color)	Set the color shown in the dialog.
set_selection([selection])	Set the channel selection.
show()	Show the dialog and execute the event loop.

Attributes

config	Get and set the config object (*UiConfig*).
title	Get / set the dialog title, it needs to be a str.
user_input	Facility to store and retrieve the user input.

property config

Get and set the config object (*UiConfig*).

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.

- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

set_color(*color: Color*) → None

Set the color shown in the dialog.

Parameters
color (*Color*) – The color to edit.

Example:

```
color_dialog.set_color( core.Color(128, 128, 128) )
```

set_selection(*selection: int = 0*)

Set the channel selection.

Parameters
selection (*int*) – The number of the channel to select (0 = red, 1 = green and 2 = blue).

Example:

```
color_dialog.set_selection(1)
```

show()

Show the dialog and execute the event loop. Until this method returns, all keyboards event are processed by the local event loop. This is also true if called from the main event loop.

This event loop returns the edited *Color* or None if the user pressed the ESC key.

Returns
The editor color.

Return type
Color

Example:

```
new_color = color_dialog.show()
```

property title

Get / set the dialog title, it needs to be a str.

property user_input

Facility to store and retrieve the user input.

ColorPicker

class pygamelib.gfx.ui.**ColorPicker**(*orientation: int = None*, *config: UiConfig = None*)

Bases: `object`

The ColorPicker widget is a simple object to select the red, green and blue components of a color.

It provides the API to set/get each color channel independently as well as the mechanism to select and draw a selection box around one specific channel to give the user a visual cue about what he is modifying.

__init__(*orientation: int = None*, *config: UiConfig = None*) → None

The constructor is really simple and takes only 2 arguments.

Parameters

- **orientation** (*Orientation*) – One of the orientation constants.
- **config** (*UiConfig*) – The configuration object.

The default orientation is horizontal.

Warning: The orientation parameter is ignored for the moment.

Example:

```
color_picker = ColorPicker(constants.Orientation.HORIZONTAL, conf)
screen.place(color_picker, 10, 10)
screen.update()
```

Methods

__init__([orientation, config])	The constructor is really simple and takes only 2 arguments.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.

Attributes

blue	Get / set the blue component of the color, the value needs to be an int between 0 and 255.
color	Get / set the edited color.
green	Get / set the green component of the color, the value needs to be an int between 0 and 255.
red	Get / set the red component of the color, the value needs to be an int between 0 and 255.
selection	Get / set the selection, it needs to be an int between 0 and 2 included.

property blue

Get / set the blue component of the color, the value needs to be an int between 0 and 255.

property color

Get / set the edited color.

The setter automatically forward the individual red, green and blue values to to the proper properties of that widget.

Parameters
value (*Color*) – The color object.

Example:

```
current_color = color_picker.color
current_color.r += 10
color_picker.color = current_color
```

property green

Get / set the green component of the color, the value needs to be an int between 0 and 255.

property red

Get / set the red component of the color, the value needs to be an int between 0 and 255.

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property selection

Get / set the selection, it needs to be an int between 0 and 2 included.

0 correspond to the red channel, 1 to the green channel and 2 to the blue channel.

When this widget is rendered a *Box* will be rendered around the specified channel.

Cursor

class pygamelib.gfx.ui.**Cursor**(*relative_row: int | None = 0*, *relative_column: int | None = 0*, *blink_time: float | None = 0.4*, *sprixel: Sprixel | None = None*, *parent: Widget | None = None*)

Bases: *PglBaseObject*

New in version 1.4.0.

The Cursor class represent a typing cursor on screen.

> **Warning:** The Cursor **need** to be rendered last! For example, in a LineInput widget, the cursor is rendered when the rest of the LineInput is already rendered . The reason being that the Cursor need to know what is already on screen in case it overlap something.

__init__(*relative_row: int | None = 0*, *relative_column: int | None = 0*, *blink_time: float | None = 0.4*, *sprixel: Sprixel | None = None*, *parent: Widget | None = None*) → None

Parameters

- **relative_row** (*int*) – The relative row position inside the parent widget.
- **relative_column** (*int*) – The relative column position inside the parent widget.
- **blink_time** (*float*) – The time interval between blinks. Default is 0.4 seconde. If you want to keep the cursor solid (i.e: not blinking) set this parameter to 0.
- **sprixel** (Sprixel) – The cursor's sprixel (or representation) as a Sprixel.
- **parent** (*Widget*) – The parent object, ie: the one in which the cursor reside. It's optional because you can share a cursor with multiple widgets.

Example:

```
# Create a cyan cursor rapidly blinking.
custom_cursor = Cursor(
    blink_time=0.1,
    sprixel=Sprixel(
        "|", bg_color=config.input_bg_color, fg_color=Color(0, 255, 255)
    ),
)
line_input = LineInput(
    "Test of the LineInput widget",
    config=UiConfig.instance(),
    minimum_width=6,
    maximum_width=200,
    cursor=custom_cursor,
)
```

Methods

__init__([relative_row, relative_column, ...])	**param relative_row** The relative row position inside the parent widget.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
lock_position()	Prevent the cursor's relative position to be changed.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
store_screen_position(row, column)	Store the screen position of the object.
unlock_position()	Authorize the cursor's relative position to be changed.

Attributes

sprixel	Get and set the sprixel of the cursor, it has to be `core.Sprixel`.
parent	Get and set the parent widget of the cursor, it has to be a *Widget* or None.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
relative_column	Get and set the relative_column of the cursor, it has to be an int.
relative_row	Get and set the relative_row of the cursor, it has to be an int.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

lock_position()

Prevent the cursor's relative position to be changed. It is useful for objects that manipulate the cursor depending on their content.

Example:

```
my_cursor = Cursor()
my_lineedit = LineEdit(cursor=my_cursor)
my_cursor.lock_position()
my_lineedit.delete()
my_cursor.unlock_position()
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property parent: *Widget* | None

Get and set the parent widget of the cursor, it has to be a *Widget* or None.

property relative_column: int

Get and set the relative_column of the cursor, it has to be an int. This value cannot be negative (as it makes no sense in our coordinate referential).

property relative_row: int

Get and set the relative_row of the cursor, it has to be an int. This value cannot be negative (as it makes no sense in our coordinate referential).

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

property sprixel: *Sprixel*

Get and set the sprixel of the cursor, it has to be `core.Sprixel`.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

unlock_position() → None

Authorize the cursor's relative position to be changed. It is useful for objects that manipulate the cursor depending on their content.

Example:

```
my_cursor = Cursor()
my_lineedit = LineEdit(cursor=my_cursor)
my_cursor.lock_position()
```

(continues on next page)

(continued from previous page)

```
my_lineedit.delete()
my_cursor.unlock_position()
```

Dialog

class pygamelib.gfx.ui.**Dialog**(*config=None*)

Bases: `object`

Dialog is a virtual class that can be subclassed to create actual dialogs.

All classes that inherits from Dialog have the following constraints:

- They need to implement a show() method.
- They are automatically rendered on the second pass by the *Screen* object.

It stores the *UiConfig* object and provide a helper attribute for user inputs.

__init__(*config=None*) → None

This constructor takes only one parameter.

Parameters

config (*UiConfig*.) – The config object.

Methods

__init__([config])	This constructor takes only one parameter.
show()	This is a virtual method, calling it directly will only raise a NotImplementedError.

Attributes

config	Get and set the config object (*UiConfig*).
user_input	Facility to store and retrieve the user input.

property config

Get and set the config object (*UiConfig*).

show()

This is a virtual method, calling it directly will only raise a NotImplementedError. Each class that inheritate Dialog needs to implement show().

property user_input

Facility to store and retrieve the user input.

FileDialog

class pygamelib.gfx.ui.**FileDialog**(*path: Path = None*, *width: int = 20*, *height: int = 10*, *title: str = 'File dialog'*, *show_hidden_files: bool = False*, *filter: str = '*'*, *config: UiConfig = None*)

Bases: *Dialog*

The FileDialog is a file selection dialog: it allow the user to select a file on disk in a relatively easy way. File can then be use for any purpose by the program, like for "save as" or "open" features.

The show() method returns the path selected by the user.

Key mapping:

- ESC: set the path to None and exit from the *show()* method.
- ENTER: Exit from the *show()* method. Returns the currently selected path.
- BACKSPACE / DELETE: delete a character (both keys have the same result).
- UP / DOWN: Navigate between the files.
- LEFT / RIGHT: Navigate between the directories.
- All other keys input characters in the input field.

In all cases, when the dialog is closed, a path is returned. It can be a file name entered by the user or an existing file. The returned value can also be None if the user pressed ESC. There is no guarantee that the returned path is correct. Please, check it before doing anything with it.

Like all dialogs, it is automatically destroyed on exit of the *show()* method. It is also deleted from the screen buffer.

__init__(*path: Path = None*, *width: int = 20*, *height: int = 10*, *title: str = 'File dialog'*, *show_hidden_files: bool = False*, *filter: str = '*'*, *config: UiConfig = None*) → None

Parameters

- **path** (pathlib.Path) – The path to start in. This path is made absolute by the constructor.
- **width** (*int*) – The width of the file dialog widget (in number of screen cells).
- **height** (*int*) – The height of the file dialog widget (in number of screen cells).
- **title** (*str*) – The title of the dialog (written in the upper border).
- **show_hidden_files** (*bool*) – Does the file dialog needs to show the hidden files or not.
- **filter** (*str*) – A string that will be used to filter the files shown to the user. For example "*.spr".
- **config** (*UiConfig*) – The configuration object.

Example:

```
file_dialog = FileDialog( Path("."), 30, 10, "Open file", False, conf)
screen.place(file_dialog, 10, 10)
file = file_dialog.show()
```

Methods

__init__([path, width, height, title, ...])	**param path** The path to start in. This path is made absolute by the
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
show()	Show the dialog and execute the event loop.

Attributes

config	Get and set the config object (*UiConfig*).
filter	Get/set the current file filter.
path	Get/set the current path.
show_hidden_files	Get/set the property, if True the file dialog is going to show hidden files, and , if False, it won't.
user_input	Facility to store and retrieve the user input.

property config

Get and set the config object (*UiConfig*).

property filter

Get/set the current file filter.

Returns
The dialog's current filter.

Return type
str

property path

Get/set the current path.

Returns
The dialog's current path.

Return type
`pathlib.Path`

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.

- **width** (*int*) – The total width of the display buffer.

show() → Path

Show the dialog and execute the event loop. Until this method returns, all keyboards event are processed by the local event loop. This is also true if called from the main event loop.

This event loop returns a `pathlib.Path` object or None if the user pressed the ESC key. The path can point to an existing file or not.

Example:

```
fields = multi_input.show()
```

property show_hidden_files

Get/set the property, if True the file dialog is going to show hidden files, and , if False, it won't.

Returns
The dialog's current show_hidden_files value.

Return type
bool

property user_input

Facility to store and retrieve the user input.

FormLayout

class pygamelib.gfx.ui.**FormLayout**(*parent: Widget | None = None*)

Bases: *GridLayout*

__init__(*parent: Widget | None = None*) → None

Parameters
parent (*Widget*) – The parent object, ie: the one in which the GridLayout reside.

Example:

```
parent_widget = Widget(45, 30, config=config)
# Add a GridLayout to a widget
parent_widget.layout = GridLayout()
# You could also do:
parent_widget.layout = GridLayout(parent_widget)
# But it is not necessary because the Widget.layout property is going to do
# it for you.
```

Methods

`__init__`([parent])	**param parent** The parent object, ie: the one in which the GridLayout reside.
`add_row`(label, widget)	
`add_widget`(widget[, row, column])	Add a widget to the GridLayout.
`attach`(observer)	Attach an observer to this instance.
`count`()	Returns the amount (the count) of widgets in the GridLayout.
`count_columns`()	Returns the number of columns in the GridLayout.
`count_rows`()	Returns the number of rows in the GridLayout.
`detach`(observer)	Detach an observer from this instance.
`handle_notification`(subject[, attribute, value])	This is an implementation of the notification handling system that is necessary for this class to handle correctly widget's resizing.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`remove_row`([row])	
`render_to_buffer`(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
`store_screen_position`(row, column)	Store the screen position of the object.
`widgets`()	Returns the list of widgets that are managed by the GridLayout as a set.

Attributes

`column_minimum_width`	Get and set the column's minimum width of the layout.
`height`	Get the layout's height (including spacing).
`horizontal_spacing`	Get and set the horizontal spacing between the widgets in the layout.
`parent`	This property get/set the parent of the Layout (if any).
`row_minimum_height`	Get and set the row's minimum width of the layout.
`screen_column`	A property to get/set the screen column.
`screen_row`	A property to get/set the screen row.
`spacing`	Get and set the spacing between the widgets in the layout.
`vertical_spacing`	Get and set the vertical spacing between the widgets in the layout.
`width`	Get the layout's width (including spacing).

add_row(*label:* Text, *widget:* Widget) → int

add_widget(*widget:* Widget, *row: int = None*, *column: int = None*) → bool

Add a widget to the GridLayout. If the widget is correctly added to the layout this method returns True, otherwise it returns False.

Parameters

- **widget** (class:*Widget*) – The widget to add to the layout.
- **row** (*int*) – The row in the layout at which the widget should be added.
- **column** (*int*) – The column in the layout at which the widget should be added.

Example:

```
parent_widget = Widget(45, 30, config=config)
# Add a GridLayout to a widget
parent_widget.layout = GridLayout()
# Then add children widgets to the parent's layout
# Step 1: create a widget (here just a generic widget)
child_widget1 = Widget(config=UiConfig.instance())
# Step 2: add it to the layout.
parent_widget.layout.add_widget(child_widget1, 2, 3)
# That's it!
```

If either of the row or column (or both) are None, the method will find the first unused cell to put the widget in.

Important: If there's no space within the existing grid, a new line will be added. For now, the expansion policy cannot be chosen and it is vertically. In the future an expand policy could be added.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column_minimum_width: int

Get and set the column's minimum width of the layout. This will apply to all columns. It has to be an int.

count() → int

Returns the amount (the count) of widgets in the GridLayout.

count_columns() → int

Returns the number of columns in the GridLayout.

count_rows() → int

Returns the number of rows in the GridLayout.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

This is an implementation of the notification handling system that is necessary for this class to handle correctly widget's resizing. If you subclass GridLayout and you need to overload that method, please keep in mind that you need to let GridLayout.handle_notification() do its job if you want to benefit from its capabilities.

In other words, do not forget to call super().handle_notification(subject, attribute, value)!

property height: int

Get the layout's height (including spacing).

Returns an int.

This is a read-only property.

property horizontal_spacing: int

Get and set the horizontal spacing between the widgets in the layout. It has to be an int.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property parent: *Widget* | None

This property get/set the parent of the Layout (if any).

remove_row(*row: int = None*)

render_to_buffer(*buffer: numpy.array*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (`numpy.array`) – A screen buffer to render the item into.
- **row** (`int`) – The row to render in.
- **column** (`int`) – The column to render in.
- **height** (`int`) – The total height of the display buffer.
- **width** (`int`) – The total width of the display buffer.

property row_minimum_height: int

Get and set the row's minimum width of the layout. This will apply to all row. It has to be an int.

property screen_column: int

A property to get/set the screen column.

Parameters
value (`int`) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (`int`) – the screen row

Return type
int

property spacing: int

Get and set the spacing between the widgets in the layout. This property sets both the horizontal and vertical spacing. It has to be an int.

> **Warning:** if you try to retrieve the spacing and the horizontal and vertical spacing are not identical this property returns -1.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (`int`) – The row (or y) coordinate.
- **column** (`int`) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property vertical_spacing: int

Get and set the vertical spacing between the widgets in the layout. It has to be an int.

widgets() → Set[*Widget*]

Returns the list of widgets that are managed by the GridLayout as a set. This set is not guaranteed to be ordered!

property width: int

Get the layout's width (including spacing).

Returns an int.

This is a read-only property.

GridLayout

class pygamelib.gfx.ui.**GridLayout**(*parent: Widget | None = None*)

Bases: `Layout`

New in version 1.4.0.

The GridLayout is a layout to organize the widgets in a grid (shocking right?). All widgets are managed in a grid, one per cell. Layouts can be nested of course to adapt to your need.

__init__(*parent: Widget | None = None*) → None

Parameters

parent (*Widget*) – The parent object, ie: the one in which the GridLayout reside.

Example:

```
parent_widget = Widget(45, 30, config=config)
# Add a GridLayout to a widget
parent_widget.layout = GridLayout()
# You could also do:
parent_widget.layout = GridLayout(parent_widget)
# But it is not necessary because the Widget.layout property is going to do
# it for you.
```

Methods

`__init__`([parent])	**param parent** The parent object, ie: the one in which the GridLayout reside.
`add_widget`(widget[, row, column])	Add a widget to the GridLayout.
`attach`(observer)	Attach an observer to this instance.
`count`()	Returns the amount (the count) of widgets in the GridLayout.
`count_columns`()	Returns the number of columns in the GridLayout.
`count_rows`()	Returns the number of rows in the GridLayout.
`detach`(observer)	Detach an observer from this instance.
`handle_notification`(subject[, attribute, value])	This is an implementation of the notification handling system that is necessary for this class to handle correctly widget's resizing.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`render_to_buffer`(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
`store_screen_position`(row, column)	Store the screen position of the object.
`widgets`()	Returns the list of widgets that are managed by the GridLayout as a set.

Attributes

`column_minimum_width`	Get and set the column's minimum width of the layout.
`height`	Get the layout's height (including spacing).
`horizontal_spacing`	Get and set the horizontal spacing between the widgets in the layout.
`parent`	This property get/set the parent of the Layout (if any).
`row_minimum_height`	Get and set the row's minimum width of the layout.
`screen_column`	A property to get/set the screen column.
`screen_row`	A property to get/set the screen row.
`spacing`	Get and set the spacing between the widgets in the layout.
`vertical_spacing`	Get and set the vertical spacing between the widgets in the layout.
`width`	Get the layout's width (including spacing).

`add_widget`(*widget: Widget*, *row: int = None*, *column: int = None*) → bool

Add a widget to the GridLayout. If the widget is correctly added to the layout this method returns True, otherwise it returns False.

Parameters

- **`widget`** (class:*Widget*) – The widget to add to the layout.
- **`row`** (*`int`*) – The row in the layout at which the widget should be added.
- **`column`** (*`int`*) – The column in the layout at which the widget should be added.

Example:

```
parent_widget = Widget(45, 30, config=config)
# Add a GridLayout to a widget
parent_widget.layout = GridLayout()
# Then add children widgets to the parent's layout
# Step 1: create a widget (here just a generic widget)
child_widget1 = Widget(config=UiConfig.instance())
# Step 2: add it to the layout.
parent_widget.layout.add_widget(child_widget1, 2, 3)
# That's it!
```

If either of the row or column (or both) are None, the method will find the first unused cell to put the widget in.

Important: If there's no space within the existing grid, a new line will be added. For now, the expansion policy cannot be chosen and it is vertically. In the future an expand policy could be added.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column_minimum_width: int

Get and set the column's minimum width of the layout. This will apply to all columns. It has to be an int.

count() → int

Returns the amount (the count) of widgets in the GridLayout.

count_columns() → int

Returns the number of columns in the GridLayout.

count_rows() → int

Returns the number of rows in the GridLayout.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

This is an implementation of the notification handling system that is necessary for this class to handle correctly widget's resizing. If you subclass GridLayout and you need to overload that method, please keep in mind that you need to let GridLayout.handle_notification() do its job if you want to benefit from its capabilities.

In other words, do not forget to call super().handle_notification(subject, attribute, value)!

property height: int

Get the layout's height (including spacing).

Returns an int.

This is a read-only property.

property horizontal_spacing: int

Get and set the horizontal spacing between the widgets in the layout. It has to be an int.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property parent: *Widget* | None

This property get/set the parent of the Layout (if any).

render_to_buffer(*buffer: numpy.array*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.

- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property row_minimum_height: int

Get and set the row's minimum width of the layout. This will apply to all row. It has to be an int.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

property spacing: int

Get and set the spacing between the widgets in the layout. This property sets both the horizontal and vertical spacing. It has to be an int.

> **Warning:** if you try to retrieve the spacing and the horizontal and vertical spacing are not identical this property returns -1.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property vertical_spacing: int

Get and set the vertical spacing between the widgets in the layout. It has to be an int.

widgets() → Set[*Widget*]

Returns the list of widgets that are managed by the GridLayout as a set. This set is not guaranteed to be ordered!

property width: int

Get the layout's width (including spacing).

Returns an int.

This is a read-only property.

GridSelectorDialog

class pygamelib.gfx.ui.**GridSelectorDialog**(*choices: list = None*, *max_height: int = None*, *max_width: int = None*, *title: str = None*, *config: UiConfig = None*)

Bases: *Dialog*

The GridSelectorDialog is an easy wrapper around the *GridSelector* object. It offers a simple interface for the programmer to present a *GridSelector* to the user and retrieve its selection.

The show() method returns the path selected by the user.

Key mapping:

- ESC: set the selected item to an empty Sprixel and exit from the show() method.
- ENTER: Exit from the show() method. Returns the currently selected sprixel.
- UP / DOWN / LEFT / RIGHT: Navigate between the files.
- PAGE_UP / PAGE_DOWN: Go to previous / next page if there's any.

In all cases, when the dialog is closed, a *Sprixel* is returned.

Like all dialogs, it is automatically destroyed on exit of the *show()* method. It is also deleted from the screen buffer.

__init__(*choices: list = None*, *max_height: int = None*, *max_width: int = None*, *title: str = None*, *config: UiConfig = None*) → None

Parameters

- **choices** (*list*) – A list of choices to present to the user. The elements of the list needs to be str or *Sprixel*.
- **max_height** (*int*) – The maximum height of the grid selector.
- **max_width** (*int*) – The maximum width of the grid selector.
- **config** (*UiConfig*) – The configuration object.

Example:

```
choices = ["@","#","$","%","&","*","[","]"]
grid_dialog = GridSelector(choices, 10, 30, conf)
screen.place(grid_dialog, 10, 10)
grid_dialog.show()
```

Methods

__init__([choices, max_height, max_width, ...])	**param choices** A list of choices to present to the user. The elements of the
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
show()	Show the dialog and execute the event loop.

Attributes

config	Get and set the config object (*UiConfig*).
grid_selector	Get / set the GridSelector object, it has to be a *GridSelector* object.
title	Get / set the title of the dialog, it needs to be a str.
user_input	Facility to store and retrieve the user input.

property config

Get and set the config object (*UiConfig*).

property grid_selector

Get / set the GridSelector object, it has to be a *GridSelector* object.

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

show()

Show the dialog and execute the event loop. Until this method returns, all keyboards event are processed by the local event loop. This is also true if called from the main event loop.

This event loop returns the selected item as a *Sprixel* or None if the user pressed the ESC key.

Returns

The selected item.

Return type

Sprixel

Example:

```
item = grid_dialog.show()
```

property title

Get / set the title of the dialog, it needs to be a str.

property user_input

Facility to store and retrieve the user input.

GridSelector

class pygamelib.gfx.ui.**GridSelector**(*choices: list = None*, *max_height: int = None*, *max_width: int = None*, *config: UiConfig = None*)

Bases: `object`

The GridSelector is a widget that present a list of elements as a grid to the user.

It also provides the API to draw and manage the cursor and to retrieve the selected element.

> **Warning:** In the first version of that widget, only the characters that have a length of 1 are supported. This excludes some UTF8 characters and most of the emojis.

__init__(*choices: list = None*, *max_height: int = None*, *max_width: int = None*, *config: UiConfig = None*) → None

Parameters

- **choices** (*list*) – A list of choices to present to the user. The elements of the list needs to be str or *Sprixel*.
- **max_height** (*int*) – The maximum height of the grid selector.
- **max_width** (*int*) – The maximum width of the grid selector.
- **config** (*UiConfig*) – The configuration object.

Example:

```
choices = ["@","#","$","%","&","*","[","]"]
grid_selector = GridSelector(choices, 10, 30, conf)
screen.place(grid_selector, 10, 10)
screen.update()
```

Methods

__init__([choices, max_height, max_width, ...])	**param choices** A list of choices to present to the user. The elements of the
current_sprixel()	Returns the currently selected sprixel.
cursor_down()	Move the selection cursor one row down.
cursor_left()	Move the selection cursor one column to the left.
cursor_right()	Move the selection cursor one column to the right.
cursor_up()	Move the selection cursor one row up.
items_per_page()	Returns the number of items per page.
nb_pages()	Returns the number of pages.
page_down()	Change the current page to the one immediately down (current_page + 1).
page_up()	Change the current page to the one immediately up (current_page - 1).
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.

Attributes

choices	Get and set the list of choices, it has to be a list of *Sprixel* or str.
current_choice	Get and set the currently selected item's index (the current choice), it needs to be an int.
current_page	Get and set the current page of the grid selector, it needs to be an int.
max_height	Get and set the maximum height of the grid selector, it needs to be an int.
max_width	Get and set the maximum width of the grid selector, it needs to be an int.

property choices: int

Get and set the list of choices, it has to be a list of *Sprixel* or str.

property current_choice: int

Get and set the currently selected item's index (the current choice), it needs to be an int. Use *current_sprixel()* to get the actual current item.

property current_page: int

Get and set the current page of the grid selector, it needs to be an int.

current_sprixel() → *Sprixel*

Returns the currently selected sprixel.

cursor_down() → None

Move the selection cursor one row down.

cursor_left() → None

Move the selection cursor one column to the left.

cursor_right() → None

Move the selection cursor one column to the right.

cursor_up() → None

Move the selection cursor one row up.

items_per_page() → int

Returns the number of items per page.

property max_height: int

Get and set the maximum height of the grid selector, it needs to be an int.

property max_width: int

Get and set the maximum width of the grid selector, it needs to be an int.

nb_pages() → int

Returns the number of pages.

page_down() → None

Change the current page to the one immediately down (current_page + 1).

page_up() → None

Change the current page to the one immediately up (current_page - 1).

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

Layout

class pygamelib.gfx.ui.**Layout**(*parent:* Widget | *None = None*)

Bases: `PglBaseObject`

New in version 1.4.0.

The Layout class is mostly a virtual class. It implements a few properties but all of the methods and properties marked with the virtual method tag need to be implemented in the inheriting object.

By convention, a layout will always use the maximum space available in a rendering buffer. That means that in `render_to_buffer()` it will try to use the entire buffer_width and buffer_height while respecting the layout's constraints.

It is therefore the responsibility of the widget or layout that triggers the rendering loop to confine said layout inside its own rendering space. Most of the time it involves passing a different set of argument to `render_to_buffer()` (like the position or size of the buffer).

__init__(*parent: Widget | None = None*) → None

The Layout constructor takes the following parameters.

Parameters

parent – The parent widget. If set, it will set the parent's layout.

Methods

__init__([parent])	The Layout constructor takes the following parameters.
add_widget(w)	virtual method
attach(observer)	Attach an observer to this instance.
count()	virtual method
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render_to_buffer(buffer, row, column, ...)	virtual method
store_screen_position(row, column)	Store the screen position of the object.
widgets()	virtual method

Attributes

height	virtual attribute
parent	This property get/set the parent of the Layout (if any).
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
spacing	This property get/set the inter-widgets spacing of the Layout.
width	virtual attribute

add_widget(*w: Widget*) → bool

virtual method

This method is purely virtual and needs to be implemented in the inheriting class.

It must allow adding a *Widget* to the layout. Adding can mean different things depending on the type of layout. For example, a *GridLayout* need a row and a column to place the widget. However, these parameters are optional. All layouts should be able to add a *Widget* in the first available space.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

count() → int

virtual method

This method is purely virtual and needs to be implemented in the inheriting class.

It must count and returns as an integer the number of widgets in the layout.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property height: int

virtual attribute

This property is purely virtual and needs to be implemented in the inheriting class.

It must return the total height of the Layout.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property parent: *Widget* | None

This property get/set the parent of the Layout (if any).

render_to_buffer(*buffer: numpy.array*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

virtual method

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

This method is purely virtual and needs to be implemented in the inheriting class.

It must render the object from the display buffer to the frame buffer.

By convention, a layout will always use the maximum space available in a rendering buffer. That means that in *render_to_buffer()* it will try to use the entire buffer_width and buffer_height while respecting the layout's constraints.

It is therefore the responsibility of the widget or layout that triggers the rendering loop to confine said layout inside its own rendering space. Most of the time it involves passing a different set of argument to *render_to_buffer()* (like the position or size of the buffer).

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

property spacing: int

This property get/set the inter-widgets spacing of the Layout.

When the spacing is changed the pygamelib.gfx.ui.Layout.spacing:changed event is sent to the observers.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

widgets() → List[*Widget*]

virtual method

Returns
A list of widgets

Return type
List[*Widget*]

This method is purely virtual and needs to be implemented in the inheriting class.

It must returns a list of widgets that are contained in the layout.

property width: int

virtual attribute

This property is purely virtual and needs to be implemented in the inheriting class.

It must return the total width of the Layout.

LineInput

class pygamelib.gfx.ui.LineInput(*default: str = ''*, *filter: InputValidator = InputValidator.PRINTABLE_FILTER*, *width: int = 0*, *height: int = 0*, *minimum_width: int = 0*, *minimum_height: int = 1*, *maximum_width: int = 20*, *maximum_height: int = 1*, *config: UiConfig | None = None*, *history: History | None = None*, *cursor: Cursor | None = None*)

Bases: `Widget`

New in version 1.4.0.

The LineInput widget allows the user to enter and edit a single line of text.

This widget can be configured to accept either anything printable or only digits.

Contrary to its dialog version that widget does not have any key binding. It provides all the tools to manipulate it but it is the user's (developer) responsibility to bind keys to specific actions.

`__init__`(*default: str = ''*, *filter:* InputValidator *= InputValidator.PRINTABLE_FILTER*, *width: int = 0*, *height: int = 0*, *minimum_width: int = 0*, *minimum_height: int = 1*, *maximum_width: int = 20*, *maximum_height: int = 1*, *config:* UiConfig | *None = None*, *history:* History | *None = None*, *cursor:* Cursor | *None = None*) → None

Parameters

- **`default`** (*`str`*) – The default value in the input field.
- **`filter`** (*`InputValidator`*) – Sets the type of accepted input. It comes from the `constants` module.
- **`width`** (*`int`*) – The width of the LineInput.
- **`height`** (*`int`*) – The height of the LineInput.
- **`minimum_width`** (*`int`*) – The minimum width of the LineInput.
- **`minimum_height`** (*`int`*) – The minimum height of the LineInput.
- **`maximum_width`** (*`int`*) – The maximum width of the LineInput.
- **`maximum_height`** (*`int`*) – The maximum height of the LineInput.
- **`config`** (*`UiConfig`*) – The configuration object.
- **`history`** (*`History`*) – The history object. If none is provided, the LineInput will use the global instance of History.

Example:

```python
line_input = LineInput(
    "Test of the LineInput widget",
    config=UiConfig.instance(),
    minimum_width=6,
    maximum_width=200,
)
screen.place(line_input, 10, 10)

# Somewhere else in your code you can access the content with LineInput.text
pet_name = line_input.text
```

Methods

__init__([default, filter, width, height, ...])	**param default** The default value in the input field.
attach(observer)	Attach an observer to this instance.
backspace()	Erase the character immediately before the *Cursor*.
clear()	Clear everything from the LineInput.
cursor	A read-only property that gives access to the cursor.
delete()	Delete the character immediately under the *Cursor*.
detach(observer)	Detach an observer from this instance.
end()	Set the *Cursor*'s relative column to the length of the content (i.e: put the cursor at the end of the LineEdit).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
home()	Set the *Cursor*'s relative column to 0 (i.e: put the cursor at the beginning of the LineEdit).
insert_characters([character, position])	Insert one or more characters at a given position.
length()	Return the length of the content of the LineInput widget.
move_cursor(direction)	Move the *Cursor* in the specified direction.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
redo()	If a *History* is available, redo previously undone changes.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
store_screen_position(row, column)	Store the screen position of the object.
undo()	If a *History* is available, undo the last changes.

Attributes

bg_color	This property get/set the background color of the widget.
children	This read only property property returns the list of children widgets.
cursor	A read-only property that gives access to the cursor.
filter	Get and set the filter of the line input, it has to be an *InputValidator*.
focus	This property get/set the focus property.
height	This property get/set the height of the widget.
layout	This property get/set the layout of the widget.
maximum_height	This property get/set the maximum height of the widget.
maximum_width	This property get/set the maximum width of the widget.
minimum_height	This property get/set the minimum height of the widget.
minimum_width	This property get/set the minimum width of the widget.
parent	This property get/set the parent widget of the widget.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size_constraint	This property get/set the size constraints of the widget.
text	Get and set the text of the line input, it has to be a str.
width	This property get/set the width of the widget.
x	This property get/set the x position of the widget on screen.
y	This property get/set the y position of the widget on screen.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

backspace() → None

Erase the character immediately before the *Cursor*.

The modification is reported to the history (i.e: can be undone)

Example:

```
# If the LineInput contains "Hello"
line_input.backspace()
# Will modify it to "Hell" if the cursor is at the end of the line.
```

property bg_color: *Color*

This property get/set the background color of the widget.

When the color is changed the pygamelib.gfx.ui.Widget.bg_color:changed event is sent to the observers.

property children: Set[*Widget*]

This read only property property returns the list of children widgets.

clear() → None

Clear everything from the LineInput. If a *History* is available, it will also clear the history.

property cursor: *Cursor*

A read-only property that gives access to the cursor.

delete() → None

Delete the character immediately under the *Cursor*.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

end() → None

Set the *Cursor*'s relative column to the length of the content (i.e: put the cursor at the end of the LineEdit).

property filter: *InputValidator*

Get and set the filter of the line input, it has to be an *InputValidator*.

property focus: bool

This property get/set the focus property. It is a boolean.

At the moment it is mostly an informational property, to tell the programmer and potentially the Widget user (i.e: the class inheriting from Widget) about its own state.

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property height: int

This property get/set the height of the widget. This property respects the boundaries set by the *maximum_height* and *minimum_height* properties.

When the height is changed the pygamelib.gfx.ui.Widget.resizeEvent:height event is sent to the observers.

home() → None

Set the *Cursor*'s relative column to 0 (i.e: put the cursor at the beginning of the LineEdit).

insert_characters(*character: str = None*, *position: int | None = None*) → None

Insert one or more characters at a given position.

If no position is given, the characters are inserted at the cursor's position.

Parameters

- **character** (*str*) – The character to insert.
- **position** (*int*) – The position at which a character must be inserted.

Example:

```python
# Insert a character at position 3 of the LineInput widget (if it exists,
# otherwise insert at the end)
line_input.insert_character("a", 3)

# Insert a character at the cursor's position
line_input.insert_character("a")
```

property layout: *Layout*

This property get/set the layout of the widget. You can then add sub widgets to the layout.

This must be a *Layout* or a class that inherits from it.

When the layout is changed the pygamelib.gfx.ui.Widget.layout:changed event is sent to the observers.

length() → int

Return the length of the content of the LineInput widget.

property maximum_height: int

This property get/set the maximum height of the widget. This property is used when changing the size constraints and the height property.

property maximum_width: int

This property get/set the maximum width of the widget. This property is used when changing the size constraints and the width property.

property minimum_height: int

This property get/set the minimum height of the widget. This property is used when changing the size constraints and the height property.

property minimum_width: int

This property get/set the minimum width of the widget. This property is used when changing the size constraints and the width property.

move_cursor(*direction: Direction*) → None

Move the *Cursor* in the specified direction.

Parameters

direction (*Direction*) – The direction to move the cursor to.

Example:

```
line_edit.move_cursor(constants.Direction.LEFT)
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property parent: *Widget* | None

This property get/set the parent widget of the widget.

redo() → None

If a *History* is available, redo previously undone changes.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.

- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

property size_constraint: ***SizeConstraint***

This property get/set the size constraints of the widget. Changing the size constraints immediately resize the widget.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property text: str

Get and set the text of the line input, it has to be a str. When setting this property tries to set the LineInput.width to the size of the content if the content's length is greater than the width.

Obviously the width is constrained by the maximum_width property.

undo() → None

If a *History* is available, undo the last changes.

property width: int

This property get/set the width of the widget. This property respects the boundaries set by the *maximum_width* and *minimum_width* properties.

When the width is changed the pygamelib.gfx.ui.Widget.resizeEvent:width event is sent to the observers.

property x: int

This property get/set the x position of the widget on screen. Since a Widget is a *PglBaseObject* this is an alias for the *screen_column* property.

property y: int

This property get/set the y position of the widget on screen. Since a Widget is a *PglBaseObject* this is an alias for the *screen_row* property.

LineInputDialog

class pygamelib.gfx.ui.**LineInputDialog**(*title=None*, *label='Input a value:'*, *default=''*, *filter=InputValidator.PRINTABLE_FILTER*, *config=None*)

Bases: `Dialog`

The LineInputDialog allows the user to enter and edit a single line of text.

This dialog can be configured to accept either anything printable or only digits.

The show() method returns the user input.

Key mapping:

- ESC: set the user input to "" and exit from the `show()` method.
- ENTER: Exit from the `show()` method. Returns the user input.
- BACKSPACE / DELETE: delete a character (both keys have the same result)
- All other keys input characters in the input field.

In all cases, when the dialog is closed, the user input is returned.

Like all dialogs, it is automatically destroyed on exit of the `show()` method. It is also deleted from the screen buffer.

__init__(*title=None*, *label='Input a value:'*, *default=''*, *filter=InputValidator.PRINTABLE_FILTER*, *config=None*) → None

Parameters

- **title** (*str*) – The short title of the dialog. Only used when the dialog is not borderless.
- **label** (str | `base.Text`) – The label of the dialog (usually a one line instruction).
- **default** (*str*) – The default value in the input field.
- **filter** (*InputValidator*) – Sets the type of accepted input. It comes from the `constants` module.
- **config** (*UiConfig*) – The configuration object.

Example:

```
line_input = LineInputDialog(
    "Name the pet",
    "Enter the name of your pet:",
    "Stupido",
)
screen.place(line_input, 10, 10)
pet_name = line_input.show()
```

Methods

`__init__`([title, label, default, filter, config])	**param title** The short title of the dialog. Only used when the dialog is not
`render_to_buffer`(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
`show`()	Show the dialog and execute the event loop.

Attributes

`config`	Get and set the config object (`UiConfig`).
`label`	Get and set the label of the dialog, it has to be a str or `base.Text`.
`title`	Get and set the title of the dialog, it has to be a str.
`user_input`	Facility to store and retrieve the user input.

property config

Get and set the config object (`UiConfig`).

property label: *Text*

Get and set the label of the dialog, it has to be a str or `base.Text`.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

show()

Show the dialog and execute the event loop. Until this method returns, all keyboards event are processed by the local event loop. This is also true if called from the main event loop.

This event loop returns the either “” or what is displayed in the input field.

Example:

```
value = line_input.show()
```

property title: str

Get and set the title of the dialog, it has to be a str.

property user_input

Facility to store and retrieve the user input.

Menu

class pygamelib.gfx.ui.Menu(*title: Text = None*, *entries: list = None*, *padding: int = 1*, *config: UiConfig = None*)

Bases: `object`

The Menu object consists of a list of other Menu objects and/or *MenuAction* objects.

It has a title that is used in a *MenuBar* and the list of its entries is displayed when the menu is expanded.

A Menu object can contains an arbitrary number of entries with an arbitrary depth of submenus.

__init__(*title: Text = None*, *entries: list = None*, *padding: int = 1*, *config: UiConfig = None*) → None

The constructor takes the following parameters.

Parameters

- **title** (str | *Text*) – The title of the action (i.e: its label)
- **entries** (*list*) – A list of *MenuAction* or other Menu objects.
- **padding** (*int*) – The horizontal padding, i.e the number of space characters added to the left and right of the title.
- **config** (*UiConfig*) – The configuration object.

Example

```
menubar = MenuBar(config=UiConfig.instance(game=Game.instance()))
file_menu = Menu(
    "File",
    [
        MenuAction("Open", open_file),
        MenuAction("Save", save_file),
        MenuAction("Save as", save_file_as),
        MenuAction("Quit", exit_application),
    ]
)
menubar.add_entry( file_menu )
menubar.add_entry( MenuAction("Help", display_help) )
screen.place(menubar, 0, 0)
screen.update()
```

Methods

`__init__`([title, entries, padding, config])	The constructor takes the following parameters.
`activate`()	Activates the menu.
`add_entry`(entry)	Add an entry to the menu.
`collapse`()	Collapse the menu.
`current_entry`()	Return the currently selected menu entry.
`expand`()	Expand the menu.
`menu_width`()	Calculate and return the maximum width of the menu based on the widest element.
`render_to_buffer`(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
`select_next`()	Select the next entry in the menu.
`select_previous`()	Select the previous entry in the menu.
`title_width`()	Return the actual width of the menu title.

Attributes

`config`	Get / set the config of the Menu, it needs to be a `UiConfig`.
`entries`	Get / set the entries of the Menu, it needs to be a list of `MenuAction` objects.
`padding`	Get / set the padding before and after the menu, it needs to be an int.
`selected`	Get / set the selected status of the Menu, it needs to be a boolean.
`title`	Get / set the title of the Menu, it needs to be a `Text` object or a python str.

activate()

Activates the menu. This method contains its own event loop a bit like the show() methods of Dialogs. It expands the menu if it wasn't already the case and listen to keyboard key strokes.

- SPACE or ENTER activates (i.e execute) menu actions.
- DOWN select the next entry.
- UP select the previous entry.
- ESC or LEFT close the menu.
- RIGHT activate (i.e expand) a submenu.

Example:

```
menu.activate()
```

add_entry(*entry*)

Add an entry to the menu. An entry can be a `MenuAction` or a `Menu`. Entries are displayed in the order of there additions from left to right.

Important: The config of the entry is overwritten by the config of the Menu. That is why it's not mandatory for *Menu* and *MenuAction*.

Parameters
entry (*MenuAction* | *Menu*) – The entry to add.

Example:

```
menu.add_entry( Menu('File') )
menu.add_entry( MenuAction('Exit', quit_application) )
```

collapse()

Collapse the menu. A menu is automatically collapsed after activation.

Example:

```
file_menu.collapse()
```

property config

Get / set the config of the Menu, it needs to be a *UiConfig*.

current_entry()

Return the currently selected menu entry.

It can be either a *Menu* object or a *MenuAction* object.

property entries: list

Get / set the entries of the Menu, it needs to be a list of *MenuAction* objects.

expand()

Expand the menu. A menu is automatically expanded when activated.

Example:

```
file_menu.expand()
```

menu_width() → int

Calculate and return the maximum width of the menu based on the widest element. This includes the padding.

Returns
the menu width.

Return type
int

property padding

Get / set the padding before and after the menu, it needs to be an int.

The padding is only used when the menu is nested into another menu.

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

select_next()

Select the next entry in the menu.

The selected entry is rendered differently to give a visual feedback to the user. Please see the *UiConfig* class for the styling option available to the Menu object.

Example:

```
menu.select_next()
```

select_previous()

Select the previous entry in the menu.

The selected entry is rendered differently to give a visual feedback to the user. Please see the *UiConfig* class for the styling option available to the Menu object.

Example:

```
menu.select_previous()
```

property selected: bool

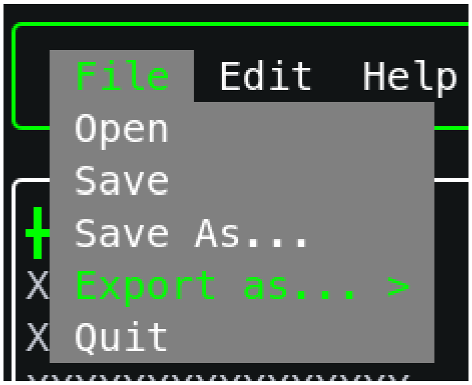
Get / set the selected status of the Menu, it needs to be a boolean.

This changes the representation (way it's drawn) of the menu entry.

property title: *Text*

Get / set the title of the Menu, it needs to be a *Text* object or a python str.

The title is used in the *MenuBar*. In the following image, the title of the expanded menu is "File".



title_width() → int

Return the actual width of the menu title. This takes into account the padding.

Example:

```
menu.title_width()
```

MenuAction

class pygamelib.gfx.ui.**MenuAction**(*title: Text = None*, *action=None*, *parameter=None*, *padding: int = 1*, *config: UiConfig = None*)

Bases: `object`

A menu action is a menu entry that executes a callback when activated. Usually a Menuaction represents an action from the user interface like open file, save, quit, etc.

Therefor a MenuAction is fairly simple, at its simplest it has a title and a callable reference to a function.

An action cannot be used by itself but can be added to a *`MenuBar`* or a *`Menu`*.

Like everything in the UI module, MenuAction are styled through a *`UiConfig`* object. Unlike the other classes of that module however, the configuration object is not mandatory when instanciating this class. The reason is that the *`MenuBar`* object impose the configuration to its managed *`MenuAction`* and *`Menu`*.

__init__(*title: Text = None*, *action=None*, *parameter=None*, *padding: int = 1*, *config: UiConfig = None*) → None

The constructor takes the following parameters.

Parameters

- **title** (str | *`Text`*) – The title of the action (i.e: its label)
- **action** (*`callable`*) – A reference to a callable function that is going to be executed when the action is activated. If set to None, nothing will happen when the action is activated.
- **parameter** (*Any*) – A parameter that is passed to the callback action if not None.
- **padding** (*`int`*) – The horizontal padding, i.e the number of space characters added to the left and right of the action.
- **config** (*`UiConfig`*) – The configuration object.

Example

```
menubar = MenuBar(config=UiConfig.instance())
file_menu = Menu(
    "File",
    [
        MenuAction("Open", open_file),
        MenuAction("Save", save_file),
        MenuAction("Save as", save_file_as),
        MenuAction("Quit", exit_application),
    ]
)
menubar.add_entry( file_menu )
menubar.add_entry( MenuAction("Help", display_help) )
screen.place(menubar, 0, 0)
screen.update()
```

Methods

__init__([title, action, parameter, ...])	The constructor takes the following parameters.
activate()	Execute and return the result of the callback.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
title_width()	Return the actual width of the action's title.

Attributes

action	Get / set the action's callback, it needs to be a callable.
config	Get / set the config of the MenuAction, it needs to be a *UiConfig*.
padding	Get / set the padding before and after the menu action, it needs to be an int.
selected	Get / set the selected of the MenuAction, it needs to be a boolean.
title	Get / set the title of the action, it needs to be a str or a *Text* object.

property action

Get / set the action's callback, it needs to be a callable.

activate()

Execute and return the result of the callback.

Example:

```
file_save_action.activate()
```

property config

Get / set the config of the MenuAction, it needs to be a *UiConfig*.

property padding

Get / set the padding before and after the menu action, it needs to be an int.

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property selected: bool

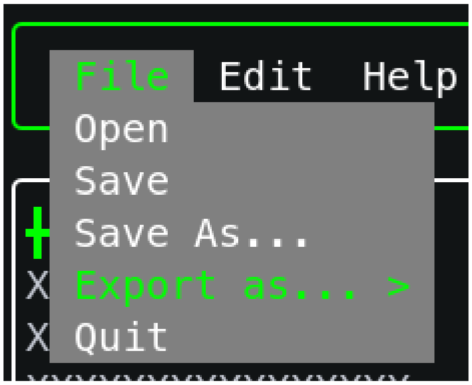
Get / set the selected of the MenuAction, it needs to be a boolean.

This changes the representation (way it's drawn) of the menu entry.

property title: *Text*

Get / set the title of the action, it needs to be a str or a `Text` object.

The title is used in the *Menu*. In the following image, the title of the first action in the expanded menu is "Open", followed by "Save".



title_width()

Return the actual width of the action's title. This takes into account the padding.

Example:

```
menu_action.title_width()
```

MenuBar

class pygamelib.gfx.ui.**MenuBar**(*entries: list = None*, *spacing: int = 2*, *config:* UiConfig *= None*)

Bases: `object`

The MenuBar widget is exactly that: an horizontal bar that can hold *Menu* or *MenuAction* objects.

Contrary to these 2 classes, MenuBar does *not* have an activate() method. The reason is that the menubar cannot block rendering with its own event loop as it is supposed to be showned at all times. So the management of interactions are left to the programmer to implement.

A typical implementation would look like this:

Example:

```
# First create a menubar
menubar = MenuBar(config=UiConfig.instance(game=Game.instance()))

# Then create a Menu
file_menu = Menu(
    "File",
    [
        MenuAction("Open", open_file),
        MenuAction("Save", save_file),
        MenuAction("Save as", save_file_as),
```

(continues on next page)

(continued from previous page)

```
        MenuAction("Quit", exit_application),
    ]
)
menubar.add_entry( file_menu )
menubar.add_entry( MenuAction("Help", display_help) )

# Place the menubar on screen
screen.place(menubar, 0, 0)
screen.update()

# Then, somewhere in an event loop, manage the inputs for example in the user
# update function
def user_update(game, inkey, elapsed_time):
    if inkey == engine.key.DOWN:
        if menubar.current_entry() is not None:
            menubar.current_entry().activate()
    elif inkey == engine.key.LEFT:
        menubar.select_previous()
    elif inkey == engine.key.RIGHT:
        menubar.select_next()
    elif inkey.name == "KEY_ENTER":
        if menubar.current_entry() is not None:
            menubar.current_entry().activate()
    elif inkey.name == "KEY_ESCAPE":
        menubar.close()
```

__init__(*entries: list = None*, *spacing: int = 2*, *config: UiConfig = None*) → None

The constructor takes the following parameters.

Parameters

- **entries** (*list*) – A list of *MenuAction* or *Menu* objects.
- **spacing** – The horizontal spacing between entries.
- **config** (*UiConfig*) – The configuration object.

Methods

__init__([entries, spacing, config])	The constructor takes the following parameters.
add_entry(entry)	Add an entry to the menu bar.
close()	Close and unselect menu entries/submenu.
current_entry()	Return the currently selected menu entry.
length()	Returns the total length of the menubar.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
select_next()	Select the next element in the menubar.
select_previous()	Select the previous element in the menubar.

Attributes

config	Get / set the config of the MenuBar, it needs to be a *UiConfig*.
current_index	Get / set the currently selected menu entry, it needs to be an int.
entries	Get / set the entries of the MenuBar, it needs to be a list of *MenuAction* or *Menu* objects.
spacing	Get / set the spacing between menu entries, it needs to be an int.

add_entry(*entry*)

Add an entry to the menu bar. An entry can be a *MenuAction* or a *Menu*. Entries are displayed in the order of there additions from left to right.

Important: The config of the entry is overwritten by the config of the MenuBar. That is why it's not mandatory for *Menu* and *MenuAction*.

Parameters
entry (*MenuAction* | *Menu*) – The entry to add.

Example:

```
menubar.add_entry( Menu('File') )
menubar.add_entry( MenuAction('Exit', quit_application) )
```

close()

Close and unselect menu entries/submenu.

Please call that method when the menu bar loses focus.

property config

Get / set the config of the MenuBar, it needs to be a *UiConfig*.

Important: The MenuBar's config is imposed on the managed items (Menu and MenuAction).

current_entry()

Return the currently selected menu entry.

It can be either a *Menu* object or a *MenuAction* object.

property current_index

Get / set the currently selected menu entry, it needs to be an int. When setting the current_index, if the previous index was corresponding to a selected entry, said entry is first unselected.

property entries: list

Get / set the entries of the MenuBar, it needs to be a list of *MenuAction* or *Menu* objects.

length() → int

Returns the total length of the menubar. This is computed everytime the method is called and it includes the spacing.

render_to_buffer(*buffer*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

select_next()

Select the next element in the menubar.

Example

```
if user_input.name == 'KEY_RIGHT':
    menubar.select_next()
```

select_previous()

Select the previous element in the menubar.

Example

```
if user_input.name == 'KEY_RIGHT':
    menubar.select_previous()
```

property spacing

Get / set the spacing between menu entries, it needs to be an int.

MessageDialog

class pygamelib.gfx.ui.**MessageDialog**(*data: list = None*, *width: int = 20*, *height: int = None*, *adaptive_height: bool = True*, *alignment: Alignment = None*, *title: str = None*, *config: UiConfig = None*)

Bases: *Dialog*

The message dialog is a popup that can display multiple lines of text.

It supports formatted text (base.Text), python strings, *pygamelib.gfx.core.Sprixel*, core.Sprite and more generally anything that can be rendered on screen (i.e: posess a render_to_buffer(self, buffer , row, column, buffer_height, buffer_width) method).

Each line can be aligned separately using one of the *Alignment* constants. Please see *add_line()*.

It also implements the *show()* virtual method of *Dialog*. This method is blocking and has its own event loop. It does not return anything.

ESC or ENTER close the dialog.

For the moment, the full message dialog needs to be displayed on screen. There is no pagination, but it is going to be implemented in a future release.

As all dialogs it also has a *user_input* property that reflects the user input. It is not used here however.

Like all dialogs, it is automatically destroyed on exit of the *show()* method. It is also deleted from the screen buffer.

Todo: Implements pagination.

__init__(*data: list = None*, *width: int = 20*, *height: int = None*, *adaptive_height: bool = True*, *alignment: Alignment = None*, *title: str = None*, *config: UiConfig = None*) → None

Parameters

- **data** (*list*) – A list of data to display inside the MessageDialog. Elements in the list can contain various data types like `base.Text`, python strings, *pygamelib.gfx.core.Sprixel*, `core.Sprite`
- **width** (*int*) – The width of the message dialog widget (in number of screen cells).
- **height** (*int*) – The height of the message dialog widget (in number of screen cells).
- **adaptive_height** (*bool*) – If True, the dialog height will be automatically adapted to match the content size.
- **alignment** (*Alignment*) – The alignment to apply to the data parameter. Please use the *Alignment* constants. The default value is *pygamelib.constants.Alignment.LEFT*
- **title** (*str*) – The short title of the dialog. Only used when the dialog is not borderless.
- **config** (*UiConfig*) – The configuration object.

Example:

```
msg = MessageDialog(
    [
        base.Text('HELP', core.Color(0,125,255), style=TextStyle.BOLD),
        base.Text('----', core.Color(0,125,255), style=TextStyle.BOLD),
        '',
    ],
    20,
    5,
    True,
    Alignment.CENTER,
)
msg.add_line('This is aligned on the right', Alignment.RIGHT)
msg.add_line('This is aligned on the left')
screen.place(msg, 10, 10)
msg.show()
```

Methods

__init__([data, width, height, ...])	**param data** A list of data to display inside the MessageDialog. Elements in
add_line(data[, alignment])	Add a line to the message dialog.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
show()	Show the dialog and execute the event loop.

Attributes

config	Get and set the config object (*UiConfig*).
height	Get and set the height of the message dialog, it has to be an int.
title	Get and set the title of the dialog, it has to be a str.
user_input	Facility to store and retrieve the user input.

add_line(*data*, *alignment: Alignment = Alignment.LEFT*) → None

Add a line to the message dialog.

The line can be any type of data that can be rendered on screen. This means that any object that expose a render_to_buffer(self, buffer, row, column, buffer_height, buffer_width) method can be added as a "line". Python strings are also obviously accepted.

Here is a non-exhaustive list of supported types:

- *Text*,
- python strings (str),
- *Sprixel*,
- *Sprite*,
- most board items,
- etc.

Parameters

- **data** (*various*) – The data to add to the message dialog.
- **alignment** (*Alignment*) – The alignment of the line to add.

Example:

```
msg.add_line(
    base.Text(
        'This is centered and very red',
        core.Color(255,0,0),
    ),
```

(continues on next page)

(continued from previous page)

```
    constants.Alignment.CENTER,
)
```

property config

Get and set the config object (*UiConfig*).

property height: int

Get and set the height of the message dialog, it has to be an int.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

show() → None

Show the dialog and execute the event loop. Until this method returns, all keyboards event are processed by the local event loop. This is also true if called from the main event loop.

This event loop returns the key pressed .

Example:

```
key_pressed = msg.show()
if key_pressed.name = 'KEY_ENTER':
    // do something
else:
    print('Good bye')
```

property title: str

Get and set the title of the dialog, it has to be a str.

property user_input

Facility to store and retrieve the user input.

MultiLineInputDialog

class pygamelib.gfx.ui.**MultiLineInputDialog**(*fields=[{'default': '', 'filter': InputValidator.PRINTABLE_FILTER, 'label': 'Input a value:'}], title: str = None, config=None*)

Bases: *Dialog*

The MultiLineInputDialog behave essentially like the *LineInputDialog* but is more configurable to allow the user to enter and edit a multiple lines of text.

Each field of this dialog can be individually configured to accept either anything printable or only digits.

The show() method returns the user input.

Key mapping:

- ESC: set the user input to “” and exit from the `show()` method.
- ENTER: Exit from the `show()` method. Returns the user input.
- BACKSPACE / DELETE: delete a character (both keys have the same result).
- TAB: cycle through the fields.
- All other keys input characters in the input field.

In all cases, when the dialog is closed, the user input is returned.

Like all dialogs, it is automatically destroyed on exit of the `show()` method. It is also deleted from the screen buffer.

`__init__`(*fields=[{'default': '', 'filter': InputValidator.PRINTABLE_FILTER, 'label': 'Input a value:'}]*, *title: str = None*, *config=None*) → None

Parameters

- **fields** (*list*) – A list of dictionnary that represent the fields to present to the user. Please see bellow for a description of the dictionnary.
- **title** (*str*) – The short title of the dialog. Only used when the dialog is not borderless.
- **config** (*UiConfig*) – The configuration object.

The fields needs to be a list that contains dictionaries. Each of the dictionaries needs to contain 3 fields:

- “label”: A one line instruction displayed over the field. This is a string.
- “default”: A string that is going to pre-fill the input field.
- “filter”: A filter to configure the acceptable inputs.

The filters needs to be a `InputValidator`.

Example:

```
fields = [
    {
        "label": "Enter the height of the new sprite:",
        "default": "",
        "filter": constants.InputValidator.INTEGER_FILTER,
    },
    {
        "label": "Enter the width of the new sprite:",
        "default": "",
        "filter": constants.InputValidator.INTEGER_FILTER,
    },
    {
        "label": "Enter the name of the new sprite:",
        "default": f"Sprite {len(sprite_list)}",
        "filter": constants.InputValidator.PRINTABLE_FILTER,
    },
]
multi_input = MultiLineInput(fields, conf)
screen.place(multi_input, 10, 10)
completed_fields = multi_input.show()
```

Methods

`__init__`([fields, title, config])	**param fields** A list of dictionnary that represent the fields to present to the
`render_to_buffer`(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
`show`()	Show the dialog and execute the event loop.

Attributes

`config`	Get and set the config object (`UiConfig`).
`fields`	Get and set the fields of the dialog, see the constructor for the format or this list.
`title`	Get and set the title of the dialog, it has to be a str.
`user_input`	Facility to store and retrieve the user input.

property config

Get and set the config object (`UiConfig`).

property fields

Get and set the fields of the dialog, see the constructor for the format or this list.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

show()

Show the dialog and execute the event loop. Until this method returns, all keyboards event are processed by the local event loop. This is also true if called from the main event loop.

This event loop returns a list of dictionaries with the content of each fields. The list of dictionaries is the same than the fields constructor parameter but each key has an additional 'user_input' field that contains the user input.

If the fields parameter was:

```
[
    {
        "label": "Input a value:",
```

(continues on next page)

(continued from previous page)

```
        "default": "",
        "filter": constants.InputValidator.PRINTABLE_FILTER,
    }
]
```

The returned value would be:

```
[
    {
        "label": "Input a value:",
        "default": "",
        "filter": constants.InputValidator.PRINTABLE_FILTER,
        "user_input": "some input",
    }
]
```

Example:

```
fields = multi_input.show()
```

property title: str

Get and set the title of the dialog, it has to be a str.

property user_input

Facility to store and retrieve the user input.

ProgressBar

class `pygamelib.gfx.ui.`**ProgressBar**(*value=0*, *maximum=100*, *width=20*, *progress_marker=''*, *empty_marker=' '*, *config=None*)

Bases: `object`

A simple horizontal progress bar widget.

__init__(*value=0*, *maximum=100*, *width=20*, *progress_marker=''*, *empty_marker=' '*, *config=None*)

Parameters

- **value** (*int*) – The initial value parameter. It represents the progression.
- **maximum** (*int*) – The maximum value held by the progress bar. Any value over the maximum is ignored.
- **width** (*int*) – The width of the progress bar widget (in number of screen cells).
- **progress_marker** (*pygamelib.gfx.core.Sprixel*) – The progress marker is displayed on progression. It is the sprixel that fills the bar. Please see below.
- **empty_marker** (*pygamelib.gfx.core.Sprixel*) – The empty marker is displayed instead of the progress marker when the bar should be empty (when the value is too low to fill the bar for example). Please see below.
- **config** (*UiConfig*) – The configuration object.

Here is a representation of were the progress and empty markers are used.

```
Progress marker
   |
[=====--------------]
           |
        Empty marker
```

Example:

```
# Create a default progress bar with the default configuration
progress_bar = ProgressBar(config=UiConfig.instance())
# Place the progress bar in the middle of the screen
screen.place(
    progress_bar, screen.vcenter, screen.hcenter - int(progress_bar.width)
)
for progress in range(progress_bar.maximum + 1):
    # Do something useful
    progress_bar.value = progress
    screen.update()
```

Methods

__init__([value, maximum, width, ...])	**param value** The initial value parameter. It represents the progression.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.

Attributes

config	Get and set the config object (*UiConfig*).
empty_marker	Get and set the empty marker, preferrably a `Sprixel` but could be a str.
maximum	Get and set the maximum possible progress, it has to be an int.
progress_marker	Get and set the progress marker, preferrably a `Sprixel` but could be a str.
value	Get and set the current progress value, it has to be an int.

property config

Get and set the config object (*UiConfig*).

property empty_marker

Get and set the empty marker, preferrably a `Sprixel` but could be a str.

property maximum

Get and set the maximum possible progress, it has to be an int.

property progress_marker

Get and set the progress marker, preferrably a `Sprixel` but could be a str.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render the object from the display buffer to the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property value

Get and set the current progress value, it has to be an int.

ProgressDialog

class pygamelib.gfx.ui.**ProgressDialog**(*label=Progress dialog[0m*, *value=0*, *maximum=100*, *width=20*, *progress_marker=''*, *empty_marker=' '*, *adaptive_width=True*, *destroy_on_complete=True*, *config=None*)

Bases: *Dialog*

ProgressDialog is a progress bar widget as a dialog (or popup). The main difference with a progress bar with borders is that it is automatically rendered on the second pass by the screen object (therefore, is visible on top of other graphical elements).

This dialog requires external interactions so it is the only dialog widget that does not provide a useful show() implementation. As a matter of fact, show do nothing at all.

ProgressDialog is mainly a label, a box and a *ProgressBar* bundled together.

__init__(*label=Progress dialog[0m*, *value=0*, *maximum=100*, *width=20*, *progress_marker=''*, *empty_marker=' '*, *adaptive_width=True*, *destroy_on_complete=True*, *config=None*)

The constructor accepts the following parameters.

Parameters

- **label** (str | `base.Text`) – A label to display on top of the progress bar.
- **value** (*int*) – The initial value parameter. It represents the progression.
- **maximum** (*int*) – The maximum value held by the progress bar. Any value over the maximum is ignored.
- **width** (*int*) – The width of the progress bar widget (in number of screen cells).
- **progress_marker** (*pygamelib.gfx.core.Sprixel*) – The progress marker is displayed on progression. It is the sprixel that fills the bar. Please see below.
- **empty_marker** (*pygamelib.gfx.core.Sprixel*) – The empty marker is displayed instead of the progress marker when the bar should be empty (when the value is too low to fill the bar for example). Please see below.

- **adaptive_width** (*bool*) – If True, the dialog will automatically adapt to the size of the label.
- **destroy_on_complete** – If True, the dialog will remove itself from the screen when complete (i.e: when value == maximum)
- **config** (*UiConfig*) – The configuration object.

Example:

```
# Create a default progress bar with the default configuration
progress_dial = ProgressDialog(
    "Please wait while I'm doing something super duper important",
    config=UiConfig.instance(),
)
# Place the progress bar in the middle of the screen
screen.place(
    progress_dial, screen.vcenter, screen.hcenter - int(progress_bar.width)
)
for progress in range(progress_dial.maximum + 1):
    # Do something useful
    progress_dial.value = progress
    screen.update()
```

Methods

__init__([label, value, maximum, width, ...])	The constructor accepts the following parameters.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
show()	The show method does nothing in the ProgressDialog.

Attributes

config	Get and set the config object (*UiConfig*).
label	Get and set the label of the dialog, it has to be a str or `base.Text`.
maximum	Get and set the maximum possible progress, it has to be an int.
user_input	Facility to store and retrieve the user input.
value	Get and set the current progress value, it has to be an int.

property config

Get and set the config object (*UiConfig*).

property label

Get and set the label of the dialog, it has to be a str or `base.Text`.

property maximum

Get and set the maximum possible progress, it has to be an int.

render_to_buffer(buffer, row, column, buffer_height, buffer_width)

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (`numpy.array`) – A screen buffer to render the item into.
- **row** (`int`) – The row to render in.
- **column** (`int`) – The column to render in.
- **height** (`int`) – The total height of the display buffer.
- **width** (`int`) – The total width of the display buffer.

show()

The show method does nothing in the ProgressDialog. It is a notable exception and the only dialog widget in the UI module to do that.

property user_input

Facility to store and retrieve the user input.

property value

Get and set the current progress value, it has to be an int.

UiConfig

class pygamelib.gfx.ui.**UiConfig**(game=None, box_vertical_border='│', box_horizontal_border='─', box_top_left_corner='', box_top_right_corner='', box_bottom_left_corner='', box_bottom_right_corner='', box_vertical_and_right='├', box_vertical_and_left='', fg_color: Color = Color(255, 255, 255), bg_color: Color = Color(0, 128, 128), fg_color_inactive: Color = Color(128, 128, 128), bg_color_selected: Color = Color(128, 128, 128), bg_color_not_selected=None, fg_color_selected: Color = Color(0, 255, 0), fg_color_not_selected: Color = Color(255, 255, 255), bg_color_menu_not_selected: Color = Color(128, 128, 128), border_fg_color: Color = Color(255, 255, 255), border_bg_color: Color = None, borderless_dialog: bool = True, widget_bg_color: Color = Color(0, 128, 128), input_fg_color: Color = Color(255, 255, 255), input_bg_color: Color = Color(163, 163, 163))

Bases: `object`

A configuration object for the UI module. TEST

This object's purpose is to configure the look and feel of the UI widgets. It does nothing by itself.

Parameters

- **game** (`Game`) – The game object.
- **box_vertical_border** (`str`) – The vertical border of a box.
- **box_horizontal_border** (`str`) – The horizontal border of a box.
- **box_top_left_corner** (`str`) – The top left corner of a box.
- **box_top_right_corner** (`str`) – The top right corner of a box.
- **box_bottom_left_corner** (`str`) – The bottom left corner of a box.

- **box_bottom_right_corner** (*str*) – The bottom right corner of a box.
- **box_vertical_and_right** (*str*) – The left junction between two boxes.
- **box_vertical_and_left** (*str*) – The right junction between two boxes.
- **fg_color** (*Color*) – The foreground color (for text and content).
- **bg_color** (*Color*) – The background color (for text and content).
- **fg_color_inactive** (*Color*) – The foreground color for inactive items like menu entries.
- **bg_color_selected** (*Color*) – The background color (for selected text and content).
- **bg_color_not_selected** (*Color*) – The background color (for non selected text and content).
- **fg_color_selected** (*Color*) – The foreground color (for selected text and content).
- **fg_color_not_selected** (*Color*) – The foreground color (for non selected text and content).
- **bg_color_menu_not_selected** (*Color*) – The menu background color (for expanded menu items).
- **border_fg_color** (*Color*) – The foreground color (for borders).
- **border_bg_color** (*Color*) – The background color (for borders).
- **borderless_dialog** (*bool*) – Is the dialog borderless or not.
- **widget_bg_color** (*Color*) – The background color of a widget.
- **input_fg_color** (*Color*) – The foreground color (i.e the text color) of a LineInput widget.
- **input_bg_color** (*Color*) – The background color of a LineInput widget.

Example:

```
config_ui_red = UiConfig(
    fg_color=Color(255,0,0),
    border_fg_color=Color(255,0,0)
)
```

__init__(game=None, box_vertical_border='│', box_horizontal_border='─', box_top_left_corner='', box_top_right_corner='', box_bottom_left_corner='', box_bottom_right_corner='', box_vertical_and_right='├', box_vertical_and_left='', fg_color: Color = Color(255, 255, 255), bg_color: Color = Color(0, 128, 128), fg_color_inactive: Color = Color(128, 128, 128), bg_color_selected: Color = Color(128, 128, 128), bg_color_not_selected=None, fg_color_selected: Color = Color(0, 255, 0), fg_color_not_selected: Color = Color(255, 255, 255), bg_color_menu_not_selected: Color = Color(128, 128, 128), border_fg_color: Color = Color(255, 255, 255), border_bg_color: Color = None, borderless_dialog: bool = True, widget_bg_color: Color = Color(0, 128, 128), input_fg_color: Color = Color(255, 255, 255), input_bg_color: Color = Color(163, 163, 163))

Methods

`__init__`([game, box_vertical_border, ...])	
`instance`(*args, **kwargs)	Returns the instance of the UiConfig object

classmethod instance(**args*, ***kwargs*)

Returns the instance of the UiConfig object

Creates an UiConfig object on first call an then returns the same instance on further calls. Useful for a default configuration. It accepts all the parameters from the constructor.

Returns

Instance of UiConfig object

Widget

class pygamelib.gfx.ui.**Widget**(*width: int = 0*, *height: int = 0*, *minimum_width: int = 0*, *minimum_height: int = 0*, *maximum_width: int = 20*, *maximum_height: int = 10*, *layout:* Layout | *None = None*, *bg_color:* Color | *None = None*, *config:* UiConfig | *None = None*)

Bases: `PglBaseObject`

New in version 1.4.0.

The Widget object is the base for all UI elements (or should be). By itself it does not do anything functionally useful. What it does however, is taking care of the geometry logic.

It enforces the geometry constraints and takes care of sending resize events messages.

__init__(*width: int = 0*, *height: int = 0*, *minimum_width: int = 0*, *minimum_height: int = 0*, *maximum_width: int = 20*, *maximum_height: int = 10*, *layout:* Layout | *None = None*, *bg_color:* Color | *None = None*, *config:* UiConfig | *None = None*) → None

Parameters

- **width** (*int*) – The width of the widget.
- **height** (*int*) – The height of the widget.
- **minimum_width** (*int*) – The minimum_width of the widget.
- **minimum_height** (*int*) – The minimum_height of the widget.
- **maximum_width** (*int*) – The maximum_width of the widget.
- **maximum_height** (*int*) – The maximum_height of the widget.
- **layout** (*Layout*) – The layout of the widget.
- **bg_color** (*Color*) – The default background color of the widget. This property overrides the *widget_bg_color* from the *UiConfig* class (in case you want to create a specific widget with a different background color than the default one).
- **config** (*UiConfig*) – The configuration object.

Example:

```
my_widget = Widget(6, 3, minimum_width=6, minimum_height=3)
my_widget.bg_color = Color(255, 255, 255)
screen.place(my_widget, 5, 2)
```

Methods

__init__([width, height, minimum_width, ...])	**param width** The width of the widget.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render_to_buffer(buffer, row, column, ...)	Render the object from the display buffer to the frame buffer.
store_screen_position(row, column)	Store the screen position of the object.

Attributes

bg_color	This property get/set the background color of the widget.
children	This read only property property returns the list of children widgets.
focus	This property get/set the focus property.
height	This property get/set the height of the widget.
layout	This property get/set the layout of the widget.
maximum_height	This property get/set the maximum height of the widget.
maximum_width	This property get/set the maximum width of the widget.
minimum_height	This property get/set the minimum height of the widget.
minimum_width	This property get/set the minimum width of the widget.
parent	This property get/set the parent widget of the widget.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
size_constraint	This property get/set the size constraints of the widget.
width	This property get/set the width of the widget.
x	This property get/set the x position of the widget on screen.
y	This property get/set the y position of the widget on screen.

`attach`(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
`observer` (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property bg_color: *Color*

This property get/set the background color of the widget.

When the color is changed the pygamelib.gfx.ui.Widget.bg_color:changed event is sent to the observers.

property children: Set[*Widget*]

This read only property property returns the list of children widgets.

`detach`(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
`observer` (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

property focus: bool

This property get/set the focus property. It is a boolean.

At the moment it is mostly an informational property, to tell the programmer and potentially the Widget user (i.e: the class inheriting from Widget) about its own state.

`handle_notification`(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property height: int

This property get/set the height of the widget. This property respects the boundaries set by the *maximum_height* and *minimum_height* properties.

When the height is changed the pygamelib.gfx.ui.Widget.resizeEvent:height event is sent to the observers.

property layout: *Layout*

This property get/set the layout of the widget. You can then add sub widgets to the layout.

This must be a *Layout* or a class that inherits from it.

When the layout is changed the pygamelib.gfx.ui.Widget.layout:changed event is sent to the observers.

property maximum_height: int

This property get/set the maximum height of the widget. This property is used when changing the size constraints and the height property.

property maximum_width: int

This property get/set the maximum width of the widget. This property is used when changing the size constraints and the width property.

property minimum_height: int

This property get/set the minimum height of the widget. This property is used when changing the size constraints and the height property.

property minimum_width: int

This property get/set the minimum width of the widget. This property is used when changing the size constraints and the width property.

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property parent: *Widget* | None

This property get/set the parent widget of the widget.

render_to_buffer(*buffer: numpy.array*, *row: int*, *column: int*, *buffer_height: int*, *buffer_width: int*) → None

Render the object from the display buffer to the frame buffer.

This method is automatically called by `pygamelib.engine.Screen.render()`.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

property size_constraint: *SizeConstraint*

This property get/set the size constraints of the widget. Changing the size constraints immediately resize the widget.

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

property width: int

This property get/set the width of the widget. This property respects the boundaries set by the *maximum_width* and *minimum_width* properties.

When the width is changed the pygamelib.gfx.ui.Widget.resizeEvent:width event is sent to the observers.

property x: int

This property get/set the x position of the widget on screen. Since a Widget is a `PglBaseObject` this is an alias for the *screen_column* property.

property y: int

This property get/set the y position of the widget on screen. Since a Widget is a `PglBaseObject` this is an alias for the *screen_row* property.

3.7.3 particles

New in version 1.3.0.

Starting with version 1.3.0, the pygamelib now provides a particle system. It is for now a first limited version and it has a number of limitations.

First, the particles are "non interactive" objects. They are not affected by board items or anything drawn on screen nor can they affect them. All particles are drawn on top of an already rendered screen.

This means no fancy particle physics out of the box. It doesn't means that it is not doable. It just means that it is not existing out of the box.

Second, although I did my best to make the particle system as efficient as possible, drawing a lot of moving elements in the terminal is very slow. So be mindful of the performances when using it.

Now despite the limitations, the particle system still allow to do some very cool stuff. Here is a video example:

This is the benchmark of the particle system, the code is available on Ghithub.

Important: Like the UI module, the particles system works exclusively with the screen buffer system (place, delete, render, update, etc.). It doesn't work with Screen functions tagged "direct display" like display_at().

CircleEmitter

class pygamelib.gfx.particles.**CircleEmitter**(*emitter_properties:* EmitterProperties *= None*)

Bases: `ParticleEmitter`

The CircleEmitter differs from the `ParticleEmitter` in only one thing: it emits its particle in a circular shape, like this:

Aside from that specificity it's exactly the same as a regular particle emitter.

__init__(*emitter_properties:* EmitterProperties *= None*) → None

The CircleEmitter takes the same parameters than the `ParticleEmitter` and make use of EmitterProperties.radius.

The radius is used as the initial distance from the center of the circle (i.e the emitter's position).

Methods

__init__([emitter_properties])	The CircleEmitter takes the same parameters than the *ParticleEmitter* and make use of EmitterProperties.radius.
apply_force(force)	Apply a force to all alive particles.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
emit([amount])	Emit a certain amount of particles.
finished()	Returns True if the emitter is finished.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a particle emitter from serialized data.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render_to_buffer(buffer, row, column, ...)	Render all the particles of that emitter in the frame buffer.
resize_pool([new_size])	In substance, this method is an alias for `ParticleEmitter.particle_pool.resize()`.
serialize()	Serialize the particle emitter.
store_screen_position(row, column)	Store the screen position of the object.
toggle_active()	Toggle the emitter's state between active and inactive.
update()	Update all the particles in the pool.

Attributes

active	Access and set the active property.
column	Access and set the column property (i.e: x).
particle_pool	This property holds this emitter's instance of a *ParticlePool*.
row	Access and set the row property (i.e: y).
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property (i.e: column).
y	Access and set the y property (i.e: row).

property active

Access and set the active property.

An emitter only emits particles if he is active. Emitted particles keeps being updated even if the emitter is not active anymore, for obvious reasons.

apply_force(*force:* Vector2D)

Apply a force to all alive particles.

The force needs to be a *Vector2D*.

Parameters

force (*Vector2D*) – The force to apply to the particles.

Example:

```
my_emitter.apply_force(base.Vector2D(0,0.3)) # slight wind.
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property (i.e: x).

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

emit(*amount: int = None*)

Emit a certain amount of particles.

The emitter will request particles from the particle pool. This in turn will trigger the recycling of dead particles if needed.

Calling this method faster than the configured emit_rate is not going to emit more particles. An emitter cannot emit particles faster than its emit_rate.

If amount is None, the emitter emits emit_number particles.

Parameters

amount (*int*) – The amount (number) of particles to be emitted.

Example:

```
my_emitter.emit(50)
```

finished()

Returns True if the emitter is finished.

A finished emitter has both:

- Reach the end of its lifespan (i.e lifespan == 0)
- And all particles are finished too.

This means that an emitter will, in most cases, not be finished as soon as its lifespan reaches 0 but a bit after. When all of its managed particles are dead.

This is on purpose for both aesthetic reasons (avoiding particles sudden removal) and for optimization (counting active particles is a O(n) operation and can be very long when there's a lot of particles so we want to do it only when necessary).

Example:

```
if my_emitter.finished():
        screen.delete(my_emitter.row, my_emitter.column)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a particle emitter from serialized data.

Parameters
data (*dict*) – The serialized data.

Returns
The loaded particle emitter.

Return type
ParticleEmitter

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.

- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property particle_pool

This property holds this emitter's instance of a *ParticlePool*.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render all the particles of that emitter in the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

resize_pool(*new_size: int = None*)

In substance, this method is an alias for `ParticleEmitter.particle_pool.resize()`. However, called without parameter, it will try to resize the particle pool to emit_number * particle_lifespan. It will do so only if the resulting number is greater than the current particle pool size.

Parameters

new_size (*int*) – The desired new size of the pool.

Example:

```
my_emitter.resize_pool(3000)
```

property row

Access and set the row property (i.e: y).

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize()

Serialize the particle emitter.

Returns
A dictionary containing all the emitter's properties.

Return type
dict

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

toggle_active()

Toggle the emitter's state between active and inactive.

An inactive emitter does not emit new particles but keeps processing particles that have already been emitted.

Example:

```
if not my_emitter.active:
    my_emitter.toggle_active()
```

update()

Update all the particles in the pool.

Updating a particle means applying particle_acceleration to every particle and then call *Particle.update()*.

Example:

```
my_emitter.update()
```

property x

Access and set the x property (i.e: column).

property y

Access and set the y property (i.e: row).

ColorParticle

class pygamelib.gfx.particles.**ColorParticle**(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *sprixel: ParticleSprixel = None*, *start_color: Color = None*, *stop_color: Color = None*)

Bases: *Particle*

This class is an extension of *Particle*. It adds the possibility to gradually go from a starting color to an end color over time. It is linked with the lifespan of the particle.

__init__(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *sprixel: ParticleSprixel = None*, *start_color: Color = None*, *stop_color: Color = None*) → None

The constructor takes the following parameters.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.
- **sprixel** (*Sprixel*) – The sprixel that represent the particle when drawn on screen.
- **start_color** (*Color*) – The color of the particle at the beginning of its lifespan.
- **stop_color** (*Color*) – The color of the particle at the end of its lifespan.

Example:

```
single_particle = ColorParticle(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
    start_color=core.Color(255, 0, 0),
    stop_color=core.Color(0, 255, 0),
)
```

Methods

__init__([row, column, velocity, lifespan, ...])	The constructor takes the following parameters.
apply_force(force)	Apply a force to the particle's acceleration vector.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
finished()	Return True if the particle is done living (i.e its lifespan is lesser or equal to 0).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a ColorParticle from a dictionary.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render([sprixel])	Render the particle as a *Sprixel*.
reset([row, column, velocity, lifespan])	Reset a particle in its initial state.
reset_lifespan([lifespan])	Reset the particle lifespan (including the initial lifespan).
serialize()	Serialize a ColorParticle into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
terminate()	Terminate a particle, i.e sets its lifespan to -1.
update()	The update method perform the calculations required to process the new particle position.

Attributes

column	Access and set the column property.
row	Access and set the row property.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property.
y	Access and set the y property.

apply_force(*force:* Vector2D) → None

Apply a force to the particle's acceleration vector.

You are more likely to apply forces to all particles of an emitter through the `apply_force()` method of the emitter class.

Parameters

force (*Vector2D*) – The force to apply.

Example:

```
gravity = Vector2D(-0.2, 0.0)
my_particle.apply_force(gravity)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

`property column`

Access and set the column property. Equivalent to the x property.

`detach`(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

`finished`() → bool

Return True if the particle is done living (i.e its lifespan is lesser or equal to 0). It returns False otherwise.

Return type
bool

Example:

```
if not my_particle.finished():
    my_particle.update()
```

`handle_notification`(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.

- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a ColorParticle from a dictionary.

Parameters

data (*dict*) – The dictionary to load from

Returns

The loaded ColorParticle

Return type

ColorParticle

Example:

```
particle = ColorParticle.load( json.load( open("particle.json") ) )
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render(*sprixel: Sprixel = None*)

Render the particle as a *Sprixel*. This method is called by the *ParticleEmitter* render_to_buffer method.

It takes a *Sprixel* as a parameter. This Sprixel is given by the ParticleEmitter.render_to_buffer() method and if it is not None, the particle will render itself into that *Sprixel* and return it.

Important: This method must be called after everything else as rendered or else there will be *Sprixel* that will be overwritten during their rendering cycle. Other elements could also have their *Sprixel* corrupted and replaced by the particle's one.

Parameters

sprixel (*Sprixel*) – A sprixel already rendered in the screen buffer.

Example:

```
p = my_particle
buffer[p.row][p.column] = p.render(buffer[p.row][p.column])
```

reset(*row: int = 0*, *column: int = 0*, *velocity:* Vector2D *= None*, *lifespan: int = None*)

Reset a particle in its initial state. This is particularly useful for the reuse of particles.

This method takes almost the same parameters than the constructor.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.

Example:

```
single_particle.reset(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

reset_lifespan(*lifespan: int = 20*) → None

Reset the particle lifespan (including the initial lifespan).

Parameters
lifespan (*int*) – The particle lifespan in number of movements/turns.

Example:

```
my_particle.reset_lifespan(10)
```

property row

Access and set the row property. Equivalent to the y property.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a ColorParticle into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( particle.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

terminate() → None

Terminate a particle, i.e sets its lifespan to -1.

In that case the ParticleEmitter and ParticlePool will recycle it. That is *IF* you are managing the particle through an emitter and/or a pool of course.

Example:

```
p = my_particle
if p.row >= screen,height or p.column >= screen.width:
    p.terminate()
```

update()

The update method perform the calculations required to process the new particle position. It mainly adds the acceleration to the velocity vector and update the position accordingly.

After calling update() the acceleration is "consumed" in the velocity and therefor reset.

The update() method takes no parameters and returns nothing.

Example:

```
my_particle.update()
```

property x

Access and set the x property. Equivalent to the column property.

property y

Access and set the y property. Equivalent to the row property.

ColorPartitionParticle

class pygamelib.gfx.particles.**ColorPartitionParticle**(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *partition: list = None*, *partition_blending_table: list = None*, *start_color: Color = None*, *stop_color: Color = None*)

Bases: `PartitionParticle`

This class is basically the same as `ColorParticle` but its base class is `PartitionParticle` instead of `Particle`. Everything else is the same.

It serves the same purpose as the `ColorParticle` with the added partition particle capabilities.

__init__(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *partition: list = None*, *partition_blending_table: list = None*, *start_color: Color = None*, *stop_color: Color = None*) → None

The constructor takes the following parameters.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.
- **partition** (*list*) – The partition of the particle.
- **partition_blending_table** (*list*) – The blending table of the particle.
- **start_color** (*Color*) – The color of the particle at the beginning of its lifespan.
- **stop_color** (*Color*) – The color of the particle at the end of its lifespan.

Example:

```
single_particle = RandomColorPartitionParticle(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

Methods

__init__([row, column, velocity, lifespan, ...])	The constructor takes the following parameters.
apply_force(force)	Apply a force to the particle's acceleration vector.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
finished()	Return True if the particle is done living (i.e its lifespan is lesser or equal to 0).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a ColorPartitionParticle from a dictionary.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render([sprixel])	This method first calls the Particle.render() method.
reset([row, column, velocity, lifespan])	Reset a particle in its initial state.
reset_lifespan([lifespan])	Reset the particle lifespan (including the initial lifespan).
serialize()	Serialize a ColorPartitionParticle into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
terminate()	Terminate a particle, i.e sets its lifespan to -1.
update()	This method first calls the Particle.update() method, then calculates the quadrant position, i.e: the actual position of the particle within a console character.

Attributes

column	Access and set the column property.
row	Access and set the row property.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property.
y	Access and set the y property.

apply_force(*force:* Vector2D) → None

Apply a force to the particle's acceleration vector.

You are more likely to apply forces to all particles of an emitter through the `apply_force()` method of the emitter class.

Parameters

force (*Vector2D*) – The force to apply.

Example:

```
gravity = Vector2D(-0.2, 0.0)
my_particle.apply_force(gravity)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
`observer` (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property. Equivalent to the x property.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
`observer` (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

finished() → bool

Return True if the particle is done living (i.e its lifespan is lesser or equal to 0). It returns False otherwise.

Return type
bool

Example:

```
if not my_particle.finished():
    my_particle.update()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- `subject` (*PglBaseObject*) – The object that has changed.

- **attribute** (*str*) – The attribute that has changed, it is usually a “FQDN style” string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a ColorPartitionParticle from a dictionary.

Parameters
data (*dict*) – The dictionary to load from

Returns
The loaded ColorPartitionParticle

Return type
ColorPartitionParticle

Example:

```
particle = ColorPartitionParticle.load( json.load( open("particle.json") ) )
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render(*sprixel: Sprixel = None*)

This method first calls the Particle.render() method. Then it updates the rendered particle’s model based on the blending table.

Parameters
sprixel (*Sprixel*) – A sprixel already rendered in the screen buffer.

Example:

```
p = my_particle
buffer[p.row][p.column] = p.render(buffer[p.row][p.column])
```

reset(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*)

Reset a particle in its initial state. This is particularly useful for the reuse of particles.

This method takes almost the same parameters than the constructor.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.

- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.

Example:

```
single_particle.reset(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

reset_lifespan(*lifespan: int = 20*) → None

Reset the particle lifespan (including the initial lifespan).

Parameters
lifespan (*int*) – The particle lifespan in number of movements/turns.

Example:

```
my_particle.reset_lifespan(10)
```

property row

Access and set the row property. Equivalent to the y property.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a ColorPartitionParticle into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( particle.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

terminate() → None

Terminate a particle, i.e sets its lifespan to -1.

In that case the ParticleEmitter and ParticlePool will recycle it. That is *IF* you are managing the particle through an emitter and/or a pool of course.

Example:

```
p = my_particle
if p.row >= screen,height or p.column >= screen.width:
    p.terminate()
```

update()

This method first calls the Particle.update() method, then calculates the quadrant position, i.e: the actual position of the particle within a console character. It then updates the particle's model based on this internal position.

Example:

```
my_particle.update()
```

property x

Access and set the x property. Equivalent to the column property.

property y

Access and set the y property. Equivalent to the row property.

EmitterProperties

class pygamelib.gfx.particles.**EmitterProperties**(*row: int = 0*, *column: int = 0*, *variance: float = 2.0*, *emit_number: int = 1*, *emit_rate: float = 0.1*, *lifespan: int = 200*, *parent=None*, *particle_velocity=None*, *particle_acceleration=None*, *particle_lifespan: float = 5.0*, *radius: float = 1.0*, *particle: Particle = None*)

Bases: `object`

EmitterProperties is a class that hold configuration variables for a particle emitter. The idea is that it's easier to carry around for multiple emitters with the same configuration than multiple values in the emitter's constructor.

It holds all possible parmeters for all types of emitters. Emitters uses only the ones that they really need.

Important: In most cases these values are copied by the emitter's constructor. So changing the values during an emitter's alive cycle is not going to do anything.

Note: This class should be a @dataclass. However, support for keyword only data classes is specific to python 3.10+. So for now, it is a regular class.

`__init__`(*row: int = 0*, *column: int = 0*, *variance: float = 2.0*, *emit_number: int = 1*, *emit_rate: float = 0.1*, *lifespan: int = 200*, *parent=None*, *particle_velocity=None*, *particle_acceleration=None*, *particle_lifespan: float = 5.0*, *radius: float = 1.0*, *particle: Particle = None*) → None

Parameters

- **`row`** (*`int`*) – The row where the emitter is. It is only important for the first rendering cycle. After that, the emitter will know its position on screen.
- **`column`** (*`int`*) – The row where the emitter is. It is only important for the first rendering cycle. After that, the emitter will know its position on screen.
- **`variance`** (*`float`*) – The variance is the amount of randomness that is allowed when emitting a particle. The exact use of this parameter is specific to each emitter.
- **`emit_number`** (*`int`*) – The number of particle emitted at each timer tick.
- **`emit_rate`** (*`float`*) – The rate of emission in seconds. This value needs to be understood as "the emitter will emit **emit_number** particles every **emit_rate** seconds".
- **`lifespan`** (*`int`*) – The lifespan of the emitter in number of emission cycle. If lifespan is set to 1 for example, the emitter will only emit one burst of particles.
- **`parent`** (*`BoardItem`*) – A parent board item. If you do that manually, you will probably want to set it specifically for each emitter.
- **`particle_velocity`** (*`Vector2D`*) – The initial particle velocity. Please read the documentation of each emitter for the specific use of particle velocity.
- **`particle_acceleration`** (*`Vector2D`*) – The initial particle acceleration. Please read the documentation of each emitter for the specific use of particle acceleration.
- **`particle_lifespan`** (*`int`*) – The lifespan of the particle in number of cycles.
- **`radius`** (*`float`*) – For emitter that supports it (like the CircleEmitter), sets the radius of emission (which translate into a velocity vector for each particle).
- **`particle`** (*`Particle`*) – The particle that the emitter will emit. This can be a class reference or a fully instantiated particle. Emitters will copy it in the particle pool.

Example:

```
props = EmitterProperties(emit_number=10, emit_rate=0.1, lifespan=10)
```

Methods

__init__([row, column, variance, ...])	**param row** The row where the emitter is. It is only important for the first
load(data)	Load an EmitterProperties from a dictionary.
serialize()	Serialize an EmitterProperties into a dictionary.

classmethod load(*data*)

Load an EmitterProperties from a dictionary.

Parameters
data (*dict*) – The dictionary to load from.

Returns
The EmitterProperties object

Return type
EmitterProperties

Example:

```
emitter_properties = EmitterProperties.load(
                        json.load( open("emitter_properties.json") )
                    )
```

serialize()

Serialize an EmitterProperties into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( emitter_properties.serialize() )
```

ParticleEmitter

class pygamelib.gfx.particles.**ParticleEmitter**(*emitter_properties=None*)

Bases: *PglBaseObject*

The particle emitter is a key piece of the pygamelib's particle system: it's the part that actually do something!

The emitter takes care of managing the particles' life cycle. It emits, move, apply forces, update and draw particles on screen. It also provide convenient methods to manage the particle pool or apply forces to all active particles in the pool.

Particle emitters are configured with *EmitterProperties*. This is a convenient way to place multiple emitters with the same configuration. For example, if you create a "torch fire" emitter, you can use the same properties to create multiple emitters. It's less cumbersome than having the parameters tied to an instance of the emitter.

Here is an example of that taken from examples/benchmark-particle-system:

Example:

```
# The torch fire properties
emt_props = particles.EmitterProperties(
    screen.vcenter, # Position is not important as it will be updated by the
    screen.hcenter, # ParticleEmitter.render_to_buffer method.
    lifespan=150,
    variance=0.3,
    emit_number=10,
    emit_rate=0.1,
    particle=particles.ColorPartitionParticle(
        start_color=core.Color(45, 151, 227),
        stop_color=core.Color(7, 2, 40),
    ),
    particle_lifespan=5,
    radius=0.4,
)
# Now create multiple emitters at different position with the same properties.
for c in [[20, 24], [20, 35], [20, 122], [20, 133]]:
    bench_state.particle_emitters.append(particles.CircleEmitter(emt_props))
    screen.place(
        bench_state.particle_emitters[-1],
        screen.vcenter - int(bench_state.altar_sprite.height / 2) + c[0],
        c[1],
        2, # Always set your emitters to be rendered on the second pass.
    )
```

Important: The entire particle system is build around the **Screen Buffer** system and is completely incompatible with the direct display system. If you want to use the particle system you **have to** use Screen.place() and the other methods of the **Screen Buffer** system.

An emitter should always be placed on screen and set to render on the second rendering pass.

It is important if you want to avoid artifacts (like particles being rendered only under the position of the emitter).

The particles by themselves are not able to render on screen, the emitter is doing that job for them.

It also means that the particles are rendered and displayed over a screen that is already rendered. Therefor, by default and for the moment, they cannot interact with elements on screen or items in a board. It also means that there is no built in particle physics (for the moment).

`__init__`(*emitter_properties=None*) → None

The constructor takes the following parameter:

Parameters

`emitter_properties` (*`EmitterProperties`*) – The properties of that particle emitter.

Methods

`__init__`([emitter_properties])	The constructor takes the following parameter:
`apply_force`(force)	Apply a force to all alive particles.
`attach`(observer)	Attach an observer to this instance.
`detach`(observer)	Detach an observer from this instance.
`emit`([amount])	Emit a certain amount of particles.
`finished`()	Returns True if the emitter is finished.
`handle_notification`(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
`load`(data)	Load a particle emitter from serialized data.
`notify`([modifier, attribute, value])	Notify all the observers that a change occurred.
`render_to_buffer`(buffer, row, column, ...)	Render all the particles of that emitter in the frame buffer.
`resize_pool`([new_size])	In substance, this method is an alias for `ParticleEmitter.particle_pool.resize()`.
`serialize`()	Serialize the particle emitter.
`store_screen_position`(row, column)	Store the screen position of the object.
`toggle_active`()	Toggle the emitter's state between active and inactive.
`update`()	Update all the particles in the pool.

Attributes

`active`	Access and set the active property.
`column`	Access and set the column property (i.e: x).
`particle_pool`	This property holds this emitter's instance of a `ParticlePool`.
`row`	Access and set the row property (i.e: y).
`screen_column`	A property to get/set the screen column.
`screen_row`	A property to get/set the screen row.
`x`	Access and set the x property (i.e: column).
`y`	Access and set the y property (i.e: row).

property active

Access and set the active property.

An emitter only emits particles if he is active. Emitted particles keeps being updated even if the emitter is not active anymore, for obvious reasons.

apply_force(*force:* Vector2D)

Apply a force to all alive particles.

The force needs to be a `Vector2D`.

Parameters

force (`Vector2D`) – The force to apply to the particles.

Example:

```
my_emitter.apply_force(base.Vector2D(0,0.3)) # slight wind.
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property (i.e: x).

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

emit(*amount: int = None*) → None

Emit a certain amount of particles.

The emitter will request particles from the particle pool. This in turn will trigger the recycling of dead particles if needed.

Calling this method faster than the configured emit_rate is not going to emit more particles. An emitter cannot emit particles faster than its emit_rate.

If amount is None, the emitter emits emit_number particles.

Parameters

amount (*int*) – The amount (number) of particles to be emitted.

Example:

```
my_emitter.emit(50)
```

finished()

Returns True if the emitter is finished.

A finished emitter has both:

- Reach the end of its lifespan (i.e lifespan == 0)
- And all particles are finished too.

This means that an emitter will, in most cases, not be finished as soon as its lifespan reaches 0 but a bit after. When all of its managed particles are dead.

This is on purpose for both aesthetic reasons (avoiding particles sudden removal) and for optimization (counting active particles is a O(n) operation and can be very long when there's a lot of particles so we want to do it only when necessary).

Example:

```
if my_emitter.finished():
        screen.delete(my_emitter.row, my_emitter.column)
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a particle emitter from serialized data.

Parameters
data (*dict*) – The serialized data.

Returns
The loaded particle emitter.

Return type
ParticleEmitter

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

property particle_pool

This property holds this emitter's instance of a *ParticlePool*.

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render all the particles of that emitter in the frame buffer.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

resize_pool(*new_size: int = None*)

In substance, this method is an alias for `ParticleEmitter.particle_pool.resize()`. However, called without parameter, it will try to resize the particle pool to emit_number * particle_lifespan. It will do so only if the resulting number is greater than the current particle pool size.

Parameters

new_size (*int*) – The desired new size of the pool.

Example:

```
my_emitter.resize_pool(3000)
```

property row

Access and set the row property (i.e: y).

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters

value (*int*) – the screen row

Return type

int

serialize()

Serialize the particle emitter.

Returns

A dictionary containing all the emitter's properties.

Return type

dict

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

toggle_active()

Toggle the emitter's state between active and inactive.

An inactive emitter does not emit new particles but keeps processing particles that have already been emitted.

Example:

```
if not my_emitter.active:
    my_emitter.toggle_active()
```

update()

Update all the particles in the pool.

Updating a particle means applying particle_acceleration to every particle and then call *Particle.update()*.

Example:

```
my_emitter.update()
```

property x

Access and set the x property (i.e: column).

property y

Access and set the y property (i.e: row).

ParticlePool

class pygamelib.gfx.particles.**ParticlePool**(*size: int = None*, *emitter_properties:* EmitterProperties *= None*)

Bases: `object`

The particle pool is a structure that holds a large number of particles and make them available to the emitters.

Its main role is to optimize the performances (both speed and memory usage). It works by pre-instantiating a desired number of particles according to the *EmitterProperties* that is given to the constructor.

The particle pool is optimized to avoid searching for available particles. It sets its own size to avoid relying on anything but its last known particle made available to the emitter. So unless for specific behavior, it is probably a good idea to let it sets its own size.

It also recycle particles that are *finished()* to avoid a constant cycle of creation/destruction of a large amount of particle objects.

__init__(*size: int = None*, *emitter_properties:* EmitterProperties *= None*) → None

The constructor takes the following parameters:

Parameters

- **size** (*int*) – The size of the pool in number of particles. For this to be efficient, be sure to have enough particles to cover for enough cycles before your first emitted particles are finished. The *ParticleEmitter* uses the following rule to size the pool: emit_rate * particle_lifespan. It is the default value if size is not specified.
- **emitter_properties** (*EmitterProperties*) – The properties of the particles that needs to be pre-instantiated.

Example:

```
my_particle_pool = ParticlePool(500, my_properties)
```

Methods

__init__([size, emitter_properties])	The constructor takes the following parameters:
count_active_particles()	Returns the number of active particle (i.e not finished) in the pool.
get_particles([amount])	Returns the requested amount of particles.
resize(new_size)	Resize the particle pool to a new size.

Attributes

pool	A read-only property that returns the particle pool tuple.

count_active_particles() → int

Returns the number of active particle (i.e not finished) in the pool.

Important: The only way to know the amount of alive particles is to go through the entire pool. Be aware of the performance impact on large particle pools.

Returns
the number of active particles.

Return type
int

Example:

```
if emitter.particles.count_active_particles() > 0:
    emitter.apply_force(gravity)
```

get_particles(*amount: int = None*) → tuple

Returns the requested amount of particles.

It is important to know that no particle is created during that call. This method returns available particles in the pool. Particles are recycled after they "died".

If amount is not specified the pool returns EmitterProperties.emit_number particles.

Parameters
amount (*int*) – The amount of particles to return.

Returns
A tuple containing the desired amount of particles.

Return type
tuple

Example:

```
fresh_particles = my_particle_pool.get_particles(30)
```

property pool: tuple

A read-only property that returns the particle pool tuple.

resize(*new_size: int*)

Resize the particle pool to a new size.

If the new size is greater than the old one, the pool will be filled by pre-instanciated particles. If it's shorter however, the extra particles will be destroyed.

Parameters
new_size (*int*) – The new size of the pool.

Example:

```
# Resize the particle pool to hold 100 particles.
my_particle_pool.resize(100)
```

Particle

class pygamelib.gfx.particles.**Particle**(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *sprixel: ParticleSprixel = None*)

Bases: *PglBaseObject*

New in version 1.3.0.

The Particle class is the base class that is inherited from by all other particles. It is mostly a “data class” in the sense that it is a class used for calculations but is not able to render on screen by itself. All operations are pure data operations until the emitter draw the particles.

Altought the Particle class can be used on its own, it is most likely to be used as a template for a particle emitter.

__init__(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *sprixel: ParticleSprixel = None*) → None

The constructor takes the following parameters.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.
- **sprixel** (*Sprixel*) – The sprixel that represent the particle when drawn on screen.

Example:

```
single_particle = Particle(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
    sprixel=core.Sprixel(graphics.GeometricShapes.BLACK_CIRCLE)
)
```

Methods

__init__([row, column, velocity, lifespan, ...])	The constructor takes the following parameters.
apply_force(force)	Apply a force to the particle's acceleration vector.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
finished()	Return True if the particle is done living (i.e its lifespan is lesser or equal to 0).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a Particle from a dictionary.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render([sprixel])	Render the particle as a *Sprixel*.
reset([row, column, velocity, lifespan])	Reset a particle in its initial state.
reset_lifespan([lifespan])	Reset the particle lifespan (including the initial lifespan).
serialize()	Serialize a Particle into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
terminate()	Terminate a particle, i.e sets its lifespan to -1.
update()	The update method perform the calculations required to process the new particle position.

Attributes

column	Access and set the column property.
row	Access and set the row property.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property.
y	Access and set the y property.

apply_force(*force:* Vector2D) → None

Apply a force to the particle's acceleration vector.

You are more likely to apply forces to all particles of an emitter through the `apply_force()` method of the emitter class.

Parameters
force (*Vector2D*) – The force to apply.

Example:

```
gravity = Vector2D(-0.2, 0.0)
my_particle.apply_force(gravity)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property. Equivalent to the x property.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

finished() → bool

Return True if the particle is done living (i.e its lifespan is lesser or equal to 0). It returns False otherwise.

Return type
bool

Example:

```
if not my_particle.finished():
    my_particle.update()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.

- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a Particle from a dictionary.

Parameters
data (*dict*) – The dictionary to load from

Returns
The loaded Particle

Return type
Particle

Example:

```
particle = Particle.load( json.load( open("particle.json") ) )
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render(*sprixel: Sprixel = None*)

Render the particle as a *Sprixel*. This method is called by the *ParticleEmitter* render_to_buffer method.

It takes a *Sprixel* as a parameter. This Sprixel is given by the ParticleEmitter.render_to_buffer() method and if it is not None, the particle will render itself into that *Sprixel* and return it.

Important: This method must be called after everything else as rendered or else there will be *Sprixel* that will be overwritten during their rendering cycle. Other elements could also have their *Sprixel* corrupted and replaced by the particle's one.

Parameters
sprixel (*Sprixel*) – A sprixel already rendered in the screen buffer.

Example:

```
p = my_particle
buffer[p.row][p.column] = p.render(buffer[p.row][p.column])
```

reset(*row: int = 0*, *column: int = 0*, *velocity:* Vector2D *= None*, *lifespan: int = None*)

Reset a particle in its initial state. This is particularly useful for the reuse of particles.

This method takes almost the same parameters than the constructor.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.

Example:

```
single_particle.reset(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

reset_lifespan(*lifespan: int = 20*) → None

Reset the particle lifespan (including the initial lifespan).

Parameters
lifespan (*int*) – The particle lifespan in number of movements/turns.

Example:

```
my_particle.reset_lifespan(10)
```

property row

Access and set the row property. Equivalent to the y property.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a Particle into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( particle.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

terminate() → None

Terminate a particle, i.e sets its lifespan to -1.

In that case the ParticleEmitter and ParticlePool will recycle it. That is *IF* you are managing the particle through an emitter and/or a pool of course.

Example:

```
p = my_particle
if p.row >= screen,height or p.column >= screen.width:
    p.terminate()
```

update() → None

The update method perform the calculations required to process the new particle position. It mainly adds the acceleration to the velocity vector and update the position accordingly.

After calling update() the acceleration is “consumed” in the velocity and therefor reset.

The update() method takes no parameters and returns nothing.

Example:

```
my_particle.update()
```

property x

Access and set the x property. Equivalent to the column property.

property y

Access and set the y property. Equivalent to the row property.

ParticleSprixel

class pygamelib.gfx.particles.**ParticleSprixel**(*model=''*, *bg_color=None*, *fg_color=None*, *is_bg_transparent=None*)

Bases: `Sprixel`

New in version 1.3.0.

The ParticleSprixel is nothing more than a `Sprixel`. Its only role is to help differentiate rendered sprixels for Partition Particles.

__init__(*model=''*, *bg_color=None*, *fg_color=None*, *is_bg_transparent=None*)

Parameters

- **model** (*str*) – The model, it can be any string. Preferrably a single character.
- **bg_color** (Color) – A Color object to configure the background color.
- **fg_color** (Color) – A Color object to configure the foreground color.
- **is_bg_transparent** (*bool*) – Set the background of the Sprixel to be transparent. It tells the engine to replace the background of the Sprixel by the background color of the overlapped sprixel.

Methods

__init__([model, bg_color, fg_color, ...])	**param model** The model, it can be any string. Preferrably a single character.
attach(observer)	Attach an observer to this instance.
black_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_RECT.
black_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_SQUARE.
blue_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_RECT.
blue_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_SQUARE.
copy()	Returns a (deep) copy of the sprixel.
cyan_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_RECT.
cyan_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_SQUARE.
detach(observer)	Detach an observer from this instance.
from_ansi(string[, model])	Takes an ANSI string, parse it and return a Sprixel.
green_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_RECT.
green_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_SQUARE.
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Create a new Sprixel object based on serialized data.
magenta_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_RECT.
magenta_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_SQUARE.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
red_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_RECT.
red_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_SQUARE.
render_to_buffer(buffer, row, column, ...)	Render the sprixel from the display buffer to the frame buffer.
serialize()	Serialize a Sprixel into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
white_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_RECT.
white_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_SQUARE.
yellow_rect()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_RECT.
yellow_square()	This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_SQUARE.

Attributes

bg_color	A property to get/set the background color of the Sprixel.
fg_color	A property to get/set the foreground color of the Sprixel.
length	Return the true length of the model.
model	A property to get/set the model of the Sprixel.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property bg_color

A property to get/set the background color of the Sprixel.

Parameters
value (Color) – The new color

When the bg_color is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprixel.bg_color:changed event. The new bg_color is passed as the *value* parameter.

Example:

```
# Access the sprixel's color
sprix.bg_color
# Set the sprixel's background color to some blue
sprix.bg_color = Color(0,128,255)
```

classmethod black_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_RECT. The difference is that BLACK_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.black_rect()
```

classmethod black_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLACK_SQUARE. The difference is that BLACK_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.black_square()
```

classmethod blue_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_RECT. The difference is that BLUE_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.blue_rect()
```

classmethod blue_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.BLUE_SQUARE. The difference is that BLUE_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.blue_square()
```

copy()

Returns a (deep) copy of the sprixel.

New in version 1.3.0.

classmethod cyan_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_RECT. The difference is that CYAN_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.cyan_rect()
```

classmethod cyan_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.CYAN_SQUARE. The difference is that CYAN_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.cyan_square()
```

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

property fg_color

A property to get/set the foreground color of the Sprixel.

Parameters
value (Color) – The new color

When the fg_color is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprixel.fg_color:changed event. The new fg_color is passed as the *value* parameter.

Example:

```
# Access the sprixel's color
sprix.fg_color
# Set the sprixel's foreground color to some green
sprix.fg_color = Color(0,255,128)
```

static from_ansi(*string*, *model=''*)

Takes an ANSI string, parse it and return a Sprixel.

Parameters

- **string** (*str*) – The ANSI string to parse.
- **model** (*str*) – The character used to represent the sprixel in the ANSI sequence. Default is “”

Example:

```
new_sprixel = Sprixel.from_ansi(
    "\x1b[48;2;139;22;19m\x1b[38;2;160;26;23m\x1b[0m"
)
```

Warning: This has mainly be tested with ANSI string generated by climage. If you find any issue, please report it

classmethod green_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_RECT. The difference is that GREEN_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.green_rect()
```

classmethod green_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.GREEN_SQUARE. The difference is that GREEN_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.green_square()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

property length

Return the true length of the model.

New in version 1.3.0.

With UTF8 and emojis the length of a string as returned by python's `len()` function is often very wrong. For example, the len("x1b[48;2;139;22;19mx1b[38;2;160;26;23mx1b[0m") returns 39 when it should return 1.

This method returns the actual printing/display size of the sprixel's model.

Note: This is a read only value. It is automatically updated when the model is changed.

Example:

```
if sprix.length > 2:
    print(
        f"Warning: that sprixel {sprix} will break the rest of the "
        "board's alignement"
        )
```

classmethod load(*data*)

Create a new Sprixel object based on serialized data.

New in version 1.3.0.

Parameters
data (*dict*) – Data loaded from JSON data (deserialized).

Return type
`Sprixel`

Example:

```
new_sprite = Sprixel.load(json_parsed_data['default_sprixel'])
```

classmethod magenta_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_RECT. The difference is that MAGENTA_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.magenta_rect()
```

classmethod magenta_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.MAGENTA_SQUARE. The difference is that MAGENTA_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.magenta_square()
```

property model

A property to get/set the model of the Sprixel.

Parameters

value (*str*) – The new model

When the model is changed, the observers are notified of the change with the pygamelib.gfx.core.Sprixel.model:changed event. The new model is passed as the *value* parameter.

Example:

```
# Get the sprixel's model
sprix.model
# Set the sprixel's model to "@"
sprix.model = "@"
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

classmethod red_rect()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_RECT. The difference is that RED_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.red_rect()
```

classmethod red_square()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.RED_SQUARE. The difference is that RED_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.red_square()
```

render_to_buffer(*buffer*, *row*, *column*, *buffer_height*, *buffer_width*)

Render the sprixel from the display buffer to the frame buffer.

New in version 1.3.0.

This method is automatically called by *pygamelib.engine.Screen.render()*.

Parameters

- **buffer** (*numpy.array*) – A screen buffer to render the item into.
- **row** (*int*) – The row to render in.
- **column** (*int*) – The column to render in.
- **height** (*int*) – The total height of the display buffer.
- **width** (*int*) – The total width of the display buffer.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a Sprixel into a dictionary.

New in version 1.3.0.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( sprixel.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

classmethod **white_rect**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_RECT. The difference is that WHITE_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.white_rect()
```

classmethod **white_square**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.WHITE_SQUARE. The difference is that WHITE_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Example:

```
sprixel = Sprixel.white_square()
```

classmethod **yellow_rect**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_RECT. The difference is that YELLOW_RECT is a string and this one is a Sprixel that can be manipulated more easily.

Note: Yellow is often rendered as brown.

Example:

```
sprixel = Sprixel.yellow_rect()
```

classmethod **yellow_square**()

This class method returns a sprixel that is the equivalent of pygamelib.assets.graphics.YELLOW_SQUARE. The difference is that YELLOW_SQUARE is a string and this one is a Sprixel that can be manipulated more easily.

Note: Yellow is often rendered as brown.

Example:

```
sprixel = Sprixel.yellow_square()
```

PartitionParticle

class pygamelib.gfx.particles.**PartitionParticle**(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *partition: list = None*, *partition_blending_table: list = None*)

Bases: *Particle*

New in version 1.3.0.

The PartitionParticle is a more precise *Particle*. Its main difference is that it is additive. This means that the PartitionParticle posess the ability to complement a sprixel that is already drawn. Or to add to a sprixel that is already drawn.

As a matter of facts, the primary goal of the PartitionParticle is to modify an already drawn sprixel to improve the visuals/graphical effects.

For example, if two particles occupy the same space on screen, with a regular *Particle* the last to render is the one that will be displayed. If one particle is represented by ‘’ and the other by ‘’, only the second will be displayed.

In the case of PartitionParticles, an addition of the 2 sprixels will be displayed! So in the previous example the addition of the 2 particles would result in ‘’ because ‘’ + ‘’ = ‘’.

It comes at a cost though as the PartitionParticle is slower to render than the *Particle* class.

The partition particle achieve that by using a partition and a blending table. The blending table is crucial for the performances to be not too catastrophic. The size of the blending table is directly linked to the performances of the PartitionParticle (the bigger the blending table the slower the rendering).

The blending table is a dictionnary of strings that covers all possible operations.

Example:

```
partition_blending_table = {
     gb.QUADRANT_UPPER_LEFT
     + gb.QUADRANT_UPPER_RIGHT: gb.UPPER_HALF_BLOCK,
     gb.QUADRANT_UPPER_LEFT + gb.QUADRANT_LOWER_LEFT: gb.LEFT_HALF_BLOCK,
     gb.QUADRANT_UPPER_LEFT
     + gb.QUADRANT_LOWER_RIGHT: gb.QUADRANT_UPPER_LEFT_AND_LOWER_RIGHT,
     # it goes on for many lines...
 }
```

By default, the PartitionParticle has a blending table that is using the UTF8 Blocks.QUADRANT_* characters. If you want to use a different one, you need to define a new blending table and pass it as parameter to the constructor.

The partition itself is a 2x2 array that contains the 4 quadrants of a character displayed in the terminal.

As an example, if a full character were a block: ‘’ the partition would be: [[’’, ‘’], [’’, ‘’]].

You can conceive the partition as the exploded version of the character/sprixel and the blending table as the rules to blend them together.

The PartitionParticle can also be used to create reinforcement effects. For example, if the partition is composed solely of ‘’ and the partition table only define one rule: ‘’ + ‘’ = ‘’. It is a powerful particle that can be used to create a lot of different effects.

Important: A limit of the current implementation is that the partition table must be a 2x2 array. It cannot be otherwise. Even if all the quadrants are the same.

`__init__`(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *partition: list = None*, *partition_blending_table: list = None*) → None

The constructor takes the following parameters.

Parameters

- **`row`** (*`int`*) – The initial row position of the particle on the screen.
- **`column`** (*`int`*) – The initial column position of the particle on the screen.
- **`velocity`** (*`Vector2D`*) – The initial velocity of the particle.
- **`lifespan`** (*`int`*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.
- **`partition`** (*`list`*) – The 2x2 array that defines the partition of the sprixel.
- **`partition_blending_table`** (*`list`*) – The blending table that defines the rules to blend the 2 sprixels.

Example:

```
# Here we'll use the default blending table
single_particle = PartitionParticle(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
    self.partition = [
        [
            graphics.Blocks.QUADRANT_UPPER_LEFT,
            graphics.Blocks.QUADRANT_UPPER_RIGHT,
        ],
        [
            graphics.Blocks.QUADRANT_LOWER_LEFT,
            graphics.Blocks.QUADRANT_LOWER_RIGHT,
        ],
    ]
)
```

Methods

__init__([row, column, velocity, lifespan, ...])	The constructor takes the following parameters.
apply_force(force)	Apply a force to the particle's acceleration vector.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
finished()	Return True if the particle is done living (i.e its lifespan is lesser or equal to 0).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a PartitionParticle from a dictionary.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render([sprixel])	This method first calls the Particle.render() method.
reset([row, column, velocity, lifespan])	Reset a particle in its initial state.
reset_lifespan([lifespan])	Reset the particle lifespan (including the initial lifespan).
serialize()	Serialize a PartitionParticle into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
terminate()	Terminate a particle, i.e sets its lifespan to -1.
update()	This method first calls the Particle.update() method, then calculates the quadrant position, i.e: the actual position of the particle within a console character.

Attributes

column	Access and set the column property.
row	Access and set the row property.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property.
y	Access and set the y property.

apply_force(*force:* Vector2D) → None

Apply a force to the particle's acceleration vector.

You are more likely to apply forces to all particles of an emitter through the `apply_force()` method of the emitter class.

Parameters

force (*Vector2D*) – The force to apply.

Example:

```
gravity = Vector2D(-0.2, 0.0)
my_particle.apply_force(gravity)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property. Equivalent to the x property.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

finished() → bool

Return True if the particle is done living (i.e its lifespan is lesser or equal to 0). It returns False otherwise.

Return type
bool

Example:

```
if not my_particle.finished():
    my_particle.update()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.

- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a PartitionParticle from a dictionary.

Parameters
data (*dict*) – The dictionary to load from

Returns
The loaded PartitionParticle

Return type
PartitionParticle

Example:

```
particle = PartitionParticle.load( json.load( open("particle.json") ) )
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render(*sprixel: Sprixel = None*)

This method first calls the Particle.render() method. Then it updates the rendered particle's model based on the blending table.

Parameters
sprixel (*Sprixel*) – A sprixel already rendered in the screen buffer.

Example:

```
p = my_particle
buffer[p.row][p.column] = p.render(buffer[p.row][p.column])
```

reset(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*)

Reset a particle in its initial state. This is particularly useful for the reuse of particles.

This method takes almost the same parameters than the constructor.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.

- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.

Example:

```
single_particle.reset(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

reset_lifespan(*lifespan: int = 20*) → None

Reset the particle lifespan (including the initial lifespan).

Parameters
lifespan (*int*) – The particle lifespan in number of movements/turns.

Example:

```
my_particle.reset_lifespan(10)
```

property row

Access and set the row property. Equivalent to the y property.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a PartitionParticle into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( particle.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

terminate() → None

Terminate a particle, i.e sets its lifespan to -1.

In that case the ParticleEmitter and ParticlePool will recycle it. That is *IF* you are managing the particle through an emitter and/or a pool of course.

Example:

```
p = my_particle
if p.row >= screen,height or p.column >= screen.width:
    p.terminate()
```

update()

This method first calls the Particle.update() method, then calculates the quadrant position, i.e: the actual position of the particle within a console character. It then updates the particle's model based on this internal position.

Example:

```
my_particle.update()
```

property x

Access and set the x property. Equivalent to the column property.

property y

Access and set the y property. Equivalent to the row property.

RandomColorParticle

class pygamelib.gfx.particles.**RandomColorParticle**(*row: int = 0*, *column: int = 0*, *velocity:* Vector2D *= None*, *lifespan: int = None*, *sprixel:* ParticleSprixel *= None*, *color:* Color *= None*)

Bases: *Particle*

This class is a *Particle* that has a random foreground color.

By default, if both the sprixel and color parameters are not specified, the model of the *Sprixel* is going to be '•' and the color will be randomly chosen.

You can also specify a color and a model.

__init__(*row: int = 0*, *column: int = 0*, *velocity:* Vector2D *= None*, *lifespan: int = None*, *sprixel:* ParticleSprixel *= None*, *color:* Color *= None*) → None

The constructor takes the following parameters.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.
- **sprixel** (*Sprixel*) – The sprixel that represent the particle when drawn on screen.
- **color** (*Color*) – The color of the particle (if you want a specific color instead of a random one).

Example:

```
single_particle = RandomColorParticle(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

Methods

__init__([row, column, velocity, lifespan, ...])	The constructor takes the following parameters.
apply_force(force)	Apply a force to the particle's acceleration vector.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
finished()	Return True if the particle is done living (i.e its lifespan is lesser or equal to 0).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a PartitionParticle from a dictionary.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render([sprixel])	Render the particle as a *Sprixel*.
reset([row, column, velocity, lifespan])	Reset a particle in its initial state.
reset_lifespan([lifespan])	Reset the particle lifespan (including the initial lifespan).
serialize()	Serialize a RandomColorParticle into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
terminate()	Terminate a particle, i.e sets its lifespan to -1.
update()	The update method perform the calculations required to process the new particle position.

Attributes

column	Access and set the column property.
row	Access and set the row property.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property.
y	Access and set the y property.

apply_force(*force:* Vector2D) → None

Apply a force to the particle's acceleration vector.

You are more likely to apply forces to all particles of an emitter through the `apply_force()` method of the emitter class.

Parameters

force (*Vector2D*) – The force to apply.

Example:

```
gravity = Vector2D(-0.2, 0.0)
my_particle.apply_force(gravity)
```

attach(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters

observer (*PglBaseObject*) – An observer to attach to this object.

Returns

True or False depending on the success of the operation.

Return type

bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property. Equivalent to the x property.

detach(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters

observer (*PglBaseObject*) – An observer to detach from this object.

Returns

True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

finished() → bool

Return True if the particle is done living (i.e its lifespan is lesser or equal to 0). It returns False otherwise.

Return type
bool

Example:

```
if not my_particle.finished():
    my_particle.update()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod **load**(*data*)

Load a PartitionParticle from a dictionary.

Parameters
data (*dict*) – The dictionary to load from

Returns
The loaded PartitionParticle

Return type
PartitionParticle

Example:

```
particle = RandomColorParticle.load( json.load( open("particle.json") ) )
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.

- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render(*sprixel: Sprixel = None*)

Render the particle as a *Sprixel*. This method is called by the *ParticleEmitter* render_to_buffer method.

It takes a *Sprixel* as a parameter. This Sprixel is given by the ParticleEmitter.render_to_buffer() method and if it is not None, the particle will render itself into that *Sprixel* and return it.

Important: This method must be called after everything else as rendered or else there will be *Sprixel* that will be overwritten during their rendering cycle. Other elements could also have their *Sprixel* corrupted and replaced by the particle's one.

Parameters

sprixel (*Sprixel*) – A sprixel already rendered in the screen buffer.

Example:

```
p = my_particle
buffer[p.row][p.column] = p.render(buffer[p.row][p.column])
```

reset(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*)

Reset a particle in its initial state. This is particularly useful for the reuse of particles.

This method takes almost the same parameters than the constructor.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.

Example:

```
single_particle.reset(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

reset_lifespan(*lifespan: int = 20*) → None

Reset the particle lifespan (including the initial lifespan).

Parameters
lifespan (*int*) – The particle lifespan in number of movements/turns.

Example:

```
my_particle.reset_lifespan(10)
```

property row

Access and set the row property. Equivalent to the y property.

property screen_column: int

A property to get/set the screen column.

Parameters
value (*int*) – the screen column

Return type
int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a RandomColorParticle into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( particle.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

terminate() → None

Terminate a particle, i.e sets its lifespan to -1.

In that case the ParticleEmitter and ParticlePool will recycle it. That is *IF* you are managing the particle through an emitter and/or a pool of course.

Example:

```
p = my_particle
if p.row >= screen,height or p.column >= screen.width:
    p.terminate()
```

update() → None

The update method perform the calculations required to process the new particle position. It mainly adds the acceleration to the velocity vector and update the position accordingly.

After calling update() the acceleration is "consumed" in the velocity and therefor reset.

The update() method takes no parameters and returns nothing.

Example:

```
my_particle.update()
```

property x

Access and set the x property. Equivalent to the column property.

property y

Access and set the y property. Equivalent to the row property.

RandomColorPartitionParticle

class pygamelib.gfx.particles.**RandomColorPartitionParticle**(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *partition: list = None*, *partition_blending_table: list = None*, *color: Color = None*)

Bases: *PartitionParticle*

This class is basically the same as *RandomColorParticle* but its base class is *PartitionParticle* instead of *Particle*. Everything else is the same.

__init__(*row: int = 0*, *column: int = 0*, *velocity: Vector2D = None*, *lifespan: int = None*, *partition: list = None*, *partition_blending_table: list = None*, *color: Color = None*) → None

The constructor takes the following parameters.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.
- **partition** (*list*) – The partition of the particle.
- **partition_blending_table** (*list*) – The blending table of the particle.

- **color** (*Color*) – The color of the particle (if you want a specific color instead of a random one).

Example:

```
single_particle = RandomColorPartitionParticle(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

Methods

__init__([row, column, velocity, lifespan, ...])	The constructor takes the following parameters.
apply_force(force)	Apply a force to the particle's acceleration vector.
attach(observer)	Attach an observer to this instance.
detach(observer)	Detach an observer from this instance.
finished()	Return True if the particle is done living (i.e its lifespan is lesser or equal to 0).
handle_notification(subject[, attribute, value])	A virtual method that needs to be implemented by the observer.
load(data)	Load a RandomColorPartitionParticle from a dictionary.
notify([modifier, attribute, value])	Notify all the observers that a change occurred.
render([sprixel])	This method first calls the Particle.render() method.
reset([row, column, velocity, lifespan])	Reset a particle in its initial state.
reset_lifespan([lifespan])	Reset the particle lifespan (including the initial lifespan).
serialize()	Serialize a RandomColorPartitionParticle into a dictionary.
store_screen_position(row, column)	Store the screen position of the object.
terminate()	Terminate a particle, i.e sets its lifespan to -1.
update()	This method first calls the Particle.update() method, then calculates the quadrant position, i.e: the actual position of the particle within a console character.

Attributes

column	Access and set the column property.
row	Access and set the row property.
screen_column	A property to get/set the screen column.
screen_row	A property to get/set the screen row.
x	Access and set the x property.
y	Access and set the y property.

apply_force(*force:* Vector2D) → None

Apply a force to the particle's acceleration vector.

You are more likely to apply forces to all particles of an emitter through the `apply_force()` method of the emitter class.

Parameters
force (*Vector2D*) – The force to apply.

Example:

```
gravity = Vector2D(-0.2, 0.0)
my_particle.apply_force(gravity)
```

`attach`(*observer*)

Attach an observer to this instance. It means that until it is detached, it will be notified every time that a notification is issued (usually on changes).

An object cannot add itself to the list of observers (to avoid infinite recursions).

Parameters
observer (*PglBaseObject*) – An observer to attach to this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
myboard = Board()
screen = Game.instance().screen
# screen will be notified of all changes in myboard
myboard.attach(screen)
```

property column

Access and set the column property. Equivalent to the x property.

`detach`(*observer*)

Detach an observer from this instance. If observer is not in the list this returns False.

Parameters
observer (*PglBaseObject*) – An observer to detach from this object.

Returns
True or False depending on the success of the operation.

Return type
bool

Example:

```
# screen will no longer be notified of the changes in myboard.
myboard.detach(screen)
```

`finished`() → bool

Return True if the particle is done living (i.e its lifespan is lesser or equal to 0). It returns False otherwise.

Return type
bool

Example:

```
if not my_particle.finished():
    my_particle.update()
```

handle_notification(*subject*, *attribute=None*, *value=None*)

A virtual method that needs to be implemented by the observer. By default it does nothing but each observer needs to implement it if something needs to be done when notified.

This method always receive the notifying object as first parameter. The 2 other parameters are optional and can be None.

You can use the attribute and value as you see fit. You are free to consider attribute as an event and value as the event's value.

Parameters

- **subject** (*PglBaseObject*) – The object that has changed.
- **attribute** (*str*) – The attribute that has changed, it is usually a "FQDN style" string. This can be None.
- **value** (*Any*) – The new value of the attribute. This can be None.

classmethod load(*data*)

Load a RandomColorPartitionParticle from a dictionary.

Parameters

data (*dict*) – The dictionary to load from

Returns

The loaded RandomColorPartitionParticle

Return type

RandomColorPartitionParticle

Example:

```
particle = RandomColorPartitionParticle.load(
                json.load( open("particle.json") )
            )
```

notify(*modifier=None*, *attribute: str = None*, *value: Any = None*) → None

Notify all the observers that a change occurred.

Parameters

- **modifier** (*PglBaseObject*) – An optional parameter that identify the modifier object to exclude it from the notified objects.
- **attribute** (*str*) – An optional parameter that identify the attribute that has changed.
- **value** (*Any*) – An optional parameter that identify the new value of the attribute.

Example:

```
# This example is silly, you would usually notify other objects from inside
# an object that changes a value that's important for the observers.
color = Color(255,200,125)
color.attach(some_text_object)
color.notify()
```

render(*sprixel: Sprixel = None*)

This method first calls the Particle.render() method. Then it updates the rendered particle's model based on the blending table.

Parameters

sprixel (*Sprixel*) – A sprixel already rendered in the screen buffer.

Example:

```
p = my_particle
buffer[p.row][p.column] = p.render(buffer[p.row][p.column])
```

reset(*row: int = 0, column: int = 0, velocity: Vector2D = None, lifespan: int = None*)

Reset a particle in its initial state. This is particularly useful for the reuse of particles.

This method takes almost the same parameters than the constructor.

Parameters

- **row** (*int*) – The initial row position of the particle on the screen.
- **column** (*int*) – The initial column position of the particle on the screen.
- **velocity** (*Vector2D*) – The initial velocity of the particle.
- **lifespan** (*int*) – The particle lifespan in number of movements/turns. A particle with a lifespan of 3 will move for 3 turns before being finished.

Example:

```
single_particle.reset(
    row=5,
    column=5,
    velocity=base.Vector2D(-0.5, 0.0),
    lifespan=10,
)
```

reset_lifespan(*lifespan: int = 20*) → None

Reset the particle lifespan (including the initial lifespan).

Parameters

lifespan (*int*) – The particle lifespan in number of movements/turns.

Example:

```
my_particle.reset_lifespan(10)
```

property row

Access and set the row property. Equivalent to the y property.

property screen_column: int

A property to get/set the screen column.

Parameters

value (*int*) – the screen column

Return type

int

property screen_row: int

A property to get/set the screen row.

Parameters
value (*int*) – the screen row

Return type
int

serialize()

Serialize a RandomColorPartitionParticle into a dictionary.

Returns
The class as a dictionary

Return type
dict

Example:

```
json.dump( particle.serialize() )
```

store_screen_position(*row: int*, *column: int*) → bool

Store the screen position of the object.

This method is automatically called by Screen.place().

Parameters

- **row** (*int*) – The row (or y) coordinate.
- **column** (*int*) – The column (or x) coordinate.

Example:

```
an_object.store_screen_coordinate(3,8)
```

terminate() → None

Terminate a particle, i.e sets its lifespan to -1.

In that case the ParticleEmitter and ParticlePool will recycle it. That is *IF* you are managing the particle through an emitter and/or a pool of course.

Example:

```
p = my_particle
if p.row >= screen,height or p.column >= screen.width:
    p.terminate()
```

update()

This method first calls the Particle.update() method, then calculates the quadrant position, i.e: the actual position of the particle within a console character. It then updates the particle's model based on this internal position.

Example:

```
my_particle.update()
```

property x

Access and set the x property. Equivalent to the column property.

property y

Access and set the y property. Equivalent to the row property.

3.8 Credits

3.8.1 Development Lead

- Arnaud Dupuis (@arnauddupuis)

3.8.2 Contributors

- Kalil de Lima (@kaozdl)
- Muhammad Syuqri (@Dansyuqri)
- Ryan Brown (@grimmjow8)
- Chase Miller (@Arekenaten)
- Gunjan Rawal (@gunjanraval)
- Anshul Choudhary (@achoudh5)
- Raymond Beaudoin (@synackray)
- Felipe Rodrigues (@fbidu)
- Bastien Wirtz (@bwirtz)
- Franz Osorio (@f-osorio)
- Guillermo Eijo (@guilleijo)
- Diego Cáceres (@diego-caceres)
- Spassarop (@spassarop)
- Javier Hernán Caballero García (@caballerojavier13)
- Olle Lögdahl (@ollelogdahl)
- MaryEtta Morris (@morrme)
- Peter Szabo (@szabopeter)
- Frans Ramirez (@Frans06)
- Krunal Rank (@KRHero03)
- Juan Picca (@jumapico)
- Harshini (@harshiniwho)
- Tammysalmon (@tammysalmon)
- JayC (@jayc13)
- Rikil Gajarla (@RikilG)
- Melsaa (@melsaa)

3.9 Release notes

3.9.1 1.3.0 (2022-10-07)

This release is massive. Please read the documentation for specific changes to classes. It is available at https://pygamelib.readthedocs.io/en/latest/index.html.

Important one: the whole pygamelib has been migrated to its own Github organization: https://github.com/pygamelib please update your links! The library's repository is now available at https://github.com/pygamelib/pygamelib.

Main updates

- **New feature:** A lot of new tools have been developed for the library and are all available on the organization's Github: https://github.com/pygamelib.
- **New feature:** The `pygamelib.engine.Screen` class now has a new **Improved Screen Management** double buffered system. This set of methods allow for a simplified management of the console screen. It is also faster than the *Legacy Direct Display* system. Please read the documentation (https://pygamelib.readthedocs.io/en/latest/pygamelib.engine.Screen.html) and the wiki on the Github repository for more about the differences. You will probably want to switch to the new stack as soon as possible. Both systems are clearly identified in the documentation by visible tags. Most of the new features of this release are **NOT** compatible with the *Legacy Direct Display* system. It still received updates and new features but will probably be deprecated in future updates.
- **New feature:** Introducing the `pygamelib.gfx.ui` module! The beginning of a module for all your game/application user interface needs. The module is in alpha for the moment, feel free to voice your feedback. This module is only compatible with the **Improved Screen Management**.
- **New feature:** A new tool has been added to the library: pgl-sprite-editor. An editor to create or edit sprites and sprite based animations.
- **New feature:** `pygamelib.engine.Game` can now be created as a Singleton through the `instance()` method.
- **New feature:** Add a particle system to the library! It includes a number of new classes that are located in the `pygamelib.gfx.particles` submodule. This module is only compatible with the **Improved Screen Management**.
- **New feature:** introducing `pygamelib.gfx.core.Font`, a Sprite based font system. This release come with an "8bits" font and a couple of font imported from FIGlet!
- **New feature:** Add a Color class (`pygamelib.gfx.core.Color`) to entirely abstract the color system.
- **New feature:** All objects can now be properly serialized and loaded through a streamlined process. Look for the *serialize()* and *load()* methods.
- **New feature:** New base object `pygamelib.base.PglBaseObject`, all objects that inherits from python's `object` are now inheriting from this new one. It implements a couple of base features but the most important is the modified *Observer* design pattern that is the base of a refactoring to event base communication within the library.
- **New feature:** Added a new board item: `pygamelib.board_item.Camera`. It is a specific item that is not shown on the board. It can be used for cinematic for example. Please read the documentation for more information.
- **New feature/improvement:** The *Board* object has been reworked to allow for a third dimension. It now has a new property called *layer*. Layers are automatically added and removed to fit the need of overlapping items. *Board.place_item()* also accept a new layer parameter to set the layer (if you want to put stuff over the player for example). An example is visible here: https://www.youtube.com/watch?v=9cOt63ZAJOk.
- *Improvement:* Most resources intensive array/list have been replaced by numpy arrays. This brings better performances for `pygamelib.engine.Board` and for `pygamelib.engine.Screen`.

- *Improvement:* Add a new algorithm to the PathFinder actuator: A*.
- *Improvement*: `pygamelib.gfx.core.Sprite` can now be tinted or modulated with a color. Both operation do the same thing: change the color of the sprite by applying a color at a given ratio. However, `tint()` returns a new sprite and does not modify the original sprite while `modulate()` returns nothing and modify the sprite directly.

Breaking changes

- `pygamelib.board_items.BoardItem` constructor parameter changed: `type` is now `item_type`.
- `pygamelib.board_items.BoardItem`: there was a conflict with `inventory_space`. It was defined both as a property and a method. The method has been removed and *BoardItem.inventory_space* is now a proper python property. Concretely: you might have to remove parenthesis when using `any_item.inventory_space` (vs the old `any_item.inventory_space()`).
- The new `pygamelib.gfx.core.Color` replaces `Terminal.on_color_rgb()` and `Terminal.color_rgb()`. It is much easier to use (just use the Color object and the pygamelib will manage foreground and background differences) but it requires to change the initialization of every Sprixel and Text of your game (sorry…).
- When using the new Improved Screen Management stack and partial display at the same time, you now have to set `Board.partial_display_focus`. It is not breaking anything in existing code but it will not behave as you want is you just `Screen.place()` your board (that uses partial display) without setting the *partial_display_focus* to the player first.

Other changes

- *Improvement*: pgl-editor now uses Sprixels instead of regular characters allowing for more possible customization and features in the Board and Screen.
- *Improvement*: in pgl-editor it is now possible to generate a random color in the color editor.
- *Improvement*: All actuators now return `pygamelib.constants.NO_DIR` if there is no direction available to `next_move()`. This makes the actuators behavior more consistent particularly when they are overloaded.
- *Improvement*: The `RandomActuator` behavior has been reworked. It now choose a direction and follow it for a certain distance before choosing a new direction. It also detect when it is stuck an, in that case, pick a new direction.
- *Improvement*: Add `display_sprite()` and `display_sprite_at()` method to Screen. These methods can display a `pygamelib.gfx.core.Sprite` on screen.
- *Improvement*: Inventory has been improved to be more versatile and less limited. It now behaves like an enhanced list of objects. A rudimentary constraints system was added (for example to limit the number of certain types of items). The new inventory is also fully plugged into the observer/notifications system.
- *Improvement*: All *BoardItem* now have configurable properties for *restorable*, *overlappable*, *pickable* and *can_move*.
- *Improvement*: `pygamelib.board_items.BoardComplexItem.sprite` is now a *@property* instead of a class variable. That property automatically call *update_sprite()*.
- *Improvement*: When `Game.mode` is set to `pygamelib.constants.MODE_RT`, all `pygamelib.board_items.Movable` now accumulate movement vectors (when using vectors). This means that non unit movement patterns are now possible.
- *Improvement*: The new `pygamelib.base.Console` implements a Singleton design pattern. You can now get a unique reference to the `blessed.Terminal` (the object wrapped in Console) object by calling *Console.instance()*.

- Fixed a bug in `pygamelib.engine.Screen.display_at()`: it was not possible to display anything after (below a Board). It is now possible.
- *Improvement*: `pygamelib.base.Text` has improved a lot. It can now use the Font system, has new attributes and is now a *PglBaseObject*. Please read the documentation for more.
- *Improvement*: Sprixels and Sprites now have their own deepcopy operator: `Sprixel.copy()` and `Sprite.copy()`.
- *Improvement*: It is now possible to set the transparency of all sprixels of a sprite by using `Sprite.set_transparency()`.
- Fixed a bug with *restorable* items: now all board items can be set to be restorable.
- Fixed a bug in pgl-editor when editing large boards that require partial display. The viewport was not correctly restored.
- Fixed issues with the library's inheritance graph.
- Fixed a bug in `pygamelib.engine.Game` where the partial display settings (when set at in the Game instance), were not correctly passed down to the Board.
- Fixed the sphinx dependencies (for building the doc).
- Fixed the mess in the sphinx files to generate the documentation.
- Fixed an issue with linting dependencies.
- Removed legacy files from older version of the library.

I would like to thank all the contributors (https://pygamelib.readthedocs.io/en/latest/authors.html) for their work on this massive update.

The new pygamelib logo was done by an awesome artist: Jack Tseng (https://hellojacktseng.carrd.co/ https://twitter.com/HelloJackTseng) please have a look at their amazing work!!

3.9.2 1.2.3 (2020-09-01)

Emergency release: fix a regression introduced by v1.2.2.

3.9.3 1.2.2 (2020-09-01)

- Fix issue with imports for Python 3.6
- Fix an issue with the way pygamelib.engine.Screen test the terminal on Windows.

3.9.4 1.2.0 (2020-08-29)

- Renamed the entire library from hac-game-lib to pygamelib.
- ***Breaking change:*** The library has been heavily refactored and this creates some issues. Please have a look at the migration notes
- **New feature:** Items that can be represented on more than one cell. We call them complex items. There's a lot of new complex items: ComplexPlayer and ComplexNPC of course, but also ComplexWall, ComplexDoor, ComplexTreasure and the general purpose Tile object.
- **New feature:** Going, with complex item we now have a proper sprite system with the gfx.core.Sprite class.

- **New feature:** In addition to the regular model we now have a new concept: the Sprixel. A Sprite is made of many Sprixels.
- **New feature:** New JSON based file format to save, load and distribute sprites and/or sprixels.
- **New feature:** All these sprites can be grouped into a SpriteCollection that in turn can be saved in our new sprite file format.
- **New feature:** New Math library. This one starts small but will grow. It makes calculating the distance and intersections easier.
- **New feature:** New Vector2D class to represent forces and movement as a vector. It is now possible to give a vector to the move() method.
- **New feature:** Gave some love to text. There are now 2 objects dedicated to text: base.Text to manipulate text and board_items.TextItem to easily place text on a board.
- **New feature:** A Screen object has been added to make the screen manipulation simpler.
- **New feature:** The Game object now has a run() method that act as the main game loop. It calls a user defined update function and takes care of a lot of things. It runs until the Game.state is set to STOPPED.
- **New feature:** The Game object can now turn by turn or real time. All movables can be configured to have time based or turn based movement speed.
- *Improvement*: The Animation class now support both regular strings (models), Sprixel and Sprite.
- *Improvement*: All complex items obviously support (actually requires) sprites but all regular board items now supports sprixels.
- *Improvement*: Test coverage dramatically improved. It has jumped from 25% to 98%.
- *Improvement*: Lots of objects now have attributes to easily access and/or set properties like position (mostly read only), width, height, etc.
- *Improvement*: Converted the editor to pygamelib and renamed it pgl-editor.py. Also added a multi page selector and integrated the new graphic assets.
- *Improvement*: All movables can now have different vertical and horizontal "steps" parameters.
- Cleaned up the repository (it was becoming seriously messy).
- Change the prefix of all exceptions from HAc to Pgl.
- Added a NO_PLAYER constant to tell the game object that he should not expect a player object.
- Improve the generated documentation.
- Various improvements in exceptions raising across the library. Please see the documentation (that was also updated).
- Various bug fixing in the Suparex example.

I also need to give some kudos to the kids of the Hyrule Astronomy Club for thorough testing of Suparex. They found well hidden bug and exploitable bugs. Special thanks to Arthur who found many glitches. Congratulations to Arthur and Hadrien that successfully exploited them to achieve extremely high scores (up to 12000!!!).

3.9.5 1.1.1 (2020-07-18)

- Fix a bug in hgl-editor: when using previously recorded parameters to create a board the editor was crashing.
- *Improvement*: Automatically enable partial display and map bigger than 40x40.
- Fix a bug a coordinates in Board.item()

3.9.6 1.1.0 (2020-06-12)

- Fix many issues with strings all across the library.
- Fix many issues with variables interpolation in exceptions.
- Fix a bug in Game.load_board() that was causing corruptions.
- Fix multiple typos in the documentation.
- Fix an issue with the user directory in hgl-editor
- Fix many issues with the PatrolActuator.
- **New feature:** partial display (dynamically display only a part of a board)
- **New feature:** new mono directional actuator.
- **New feature:** projectiles (can be sent and completely managed by the game object)
- **New feature:** new assets module to hold many non core submodules.
- **New feature:** Assets.Graphics that add thousands of glyphs (including emojis) to the current capacities of the library.
- **New feature:** Add support for PatrolActuator in hgl-editor.
- **New feature:** Add support for PathFinder actuator in hgl-editor.
- **New feature:** Add an object parent system.
- **New feature:** Add a configuration system to hgl-editor.
- *Improvement*: Add full configuration features to the Game object.
- *Improvement*: Add a new example in the form of a full procedural generation platform game (see examples/suparex).
- *Improvement*: Improved performances particularly around the features that relies on Board.place_item(). Up to 70 times faster.
- *Improvement*: It is now possible to specify the first frame index in Animation.
- *Improvement*: Formatted all the code with black.
- *Improvement*: PathFinder.add_waypoint() now sets the destination if it wasn't set before.

3.9.7 1.0.1 (2020-05-17)

- Fix a huge default save directory issue (see complete announcement) in hgl-editor.
- Fix lots of strings in hgl-editor.
- Fix a type issue in the Inventory class for the not_enough_space exception.
- Improve Board.display() performances by 15% (average).

3.9.8 1.0.0 (2020-03-20)

- Add AdvancedActuators.PathFinder @arnauddupuis
- Add test cases for BoardItem @grimmjow8 @Arekenaten
- Add test cases for Board @grimmjow8 @Arekenaten
- Add support to load files from the directories in directories.json @kaozdl
- Add a new SimpleActuators.PatrolActuator @kaozdl
- Add Animation capabilities @arnauddupuis
- Improve navigation in hgl-editor by using arrow keys @bwirtz
- Improve selection of maps in hgl-editor @gunjanraval @kaozdl
- Improve documentation for SimpleActuators.PathActuator @achoudh5
- Improve documentation for launching the test suite @bwirtz
- Migration from pip install to pipenv @kaozdl
- Fix board saving bug in hgl-editor @gunjanraval
- Fix back menu issues in hgl-editor @synackray
- Fix README and setup.py @fbidu
- Make the module compatible with Flake8: @bwirtz @arnauddupuis @kaozdl @f-osorio @guilleijo @diego-caceres @spassarop
- CircleCI integration @caballerojavier13 @bwirtz

3.9.9 2019.5

- Please see the official website.

3.9.10 pre-2019.5

- Please see the Github for history.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols