hac-game-lib Documentation

Release 1.0.1

Arnaud Dupuis

May 18, 2020

Contents:

CHAPTER 1

Board

This module contains the Board class. It is the base class for all levels.

class gamelib.Board.**Board**(***kwargs*)

A class that represent a game board.

The board is being represented by a square matrix. For the moment a board only support one player.

The Board object is the base object to build a level : you create a Board and then you add BoardItems (or objects derived from BoardItem).

Parameters

- **name** (*str*) – the name of the Board
- **size** (*list*) – array [width,height] with width and height being int. The size of the board.
- **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 *gamelib.BoardItem.BoardItemVoid*)

check_sanity()

Check the board sanity.

This is essentially an internal method called by the constructor.

clear_cell(*row*, *column*)
Clear cell (row, column)

This method clears a cell, meaning it position a void_cell BoardItemVoid at these coordinates.

Parameters

- **row** (*int*) – The row of the item to remove
- **column** (*int*) – The column of the item to remove

Example:

```
myboard.clear_cell(3,4)
```

Warning: This method does not check the content before, it *will* overwrite the content.

display()
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 is BoardItem.model. If you want to override this behavior you have to subclass BoardItem.

display_old()
Display the board.

This method display the Board (as in print()), taking care of displaying the boarders, and everything inside.

It uses the __str__ method of the item, which by default is BoardItem.model. If you want to override this behavior you have to subclass BoardItem.

get_immovables(***kwargs*)
Return a list of all the Immovable objects in the Board.

See *gamelib.Immovable.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 *gamelib.Movable.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")
```

init_board()
Initialize the board with BoardItemVoid that uses ui_board_void_cell as model.

Example:

```
myboard.init_board()
```

init_cell(*row*, *column*)
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)
```

item(*row*, *column*)
Return the item at the row, column position if within board's boundaries.

Return type *gamelib.BoardItem.BoardItem*

Raises ***HacOutOfBoardBoundException*** – if row or column are out of bound.

move(*item*, *direction*, *step*)
Move an item in the specified direction for a number of steps.

Example:

```
board.move(player,Constants.UP,1)
```

Parameters

- **item** (gamelib.Movable.Movable) – an item to move (it has to be a subclass of Movable)
- **direction** (*gamelib.Constants*) – 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 HacObjectIsNotMovableException exception (see *gamelib.HacExceptions.HacObjectIsNotMovableException*).

Important: if the move is successfull, an empty BoardItemVoid (see *gamelib.BoardItem.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.

Note: It could be interesting here, instead of relying on storing the overlapping item in a property of a Movable (*gamelib.Movable.Movable*) object, to have another dimension on the board matrix to push and pop objects on a cell. Only the first item would be rendered and it would avoid the complicated and error prone logic in this method. If anyone feel up to the challenge, PR are welcome ;-).

Todo: check all types!

place_item(*item*, *row*, *column*)

Place an item at coordinates row and column.

If row or column are our of the board boundaries, an HacOutOfBoardBoundException is raised.

If the item is not a subclass of BoardItem, an HacInvalidTypeException

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.

CHAPTER 2

BoardItem

This module contains the basic board items classes (regular and void items).

class gamelib.BoardItem.**BoardItem**(***kwargs*)

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** (*array*) – 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 '*'.

can_move()

This is a virtual method that must be implemented in deriving classes. This method has to return True or False. This represent the capacity for a BoardItem to be moved by the Board.

debug_info()

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

Return type str

display()

Print the model WITHOUT carriage return.

overlappable()

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 BoardItem to be overlapped by another BoardItem.

pickable()

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 BoardItem to be pick-up by player or NPC.

size()
This is a virtual method that must be implemented in deriving class. This method has to return an integer. This represent the size of the BoardItem. It is used for example to evaluate the space taken in the inventory.

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.BoardItem.**BoardItemVoid**(***kwargs*)
A class that represent a void cell.

overlappable()
A BoardItemVoid is obviously overlappable (so player and NPC can walk over).

Returns True

pickable()
A BoardItemVoid is not pickable, therefor this method return false.

Returns False

CHAPTER 3

Characters

This module contains the base classes for both playable and non playable characters.

class gamelib.Characters.**Character**(***kwargs*)

Bases: object

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 (*gamelib.Game*) engine.

class gamelib.Characters.**NPC**(***kwargs*)

Bases: *gamelib.Movable.Movable*, *gamelib.Characters.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:

- *gamelib.Characters.Character*
- *gamelib.Movable.Movable*
- *gamelib.BoardItem.BoardItem*

Parameters **actuator** (*gamelib.Actuators.Actuator*) – An actuator, it can be any class but it need to implement gamelibe.Actuator.Actuator.

Example:

```
mynpc = NPC(name='Idiot McStupid', type='dumb_ennemy')
mynpc.step = 1
mynpc.actuator = RandomActuator()
```

can_move()
Movable implements can_move().

Returns True

Return type Boolean

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

Return type str

display()
Print the model WITHOUT carriage return.

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 :(")
```

overlappable()
Define if the NPC is overlappable.

Obviously this method also always return False.

Returns False

Return type Boolean

Example:

```
if mynpc.overlappable():
    Utils.warn("Something is fishy, that NPC is overlappable but"
        "is not a Ghost...")
```

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...")
```

size()
This is a virtual method that must be implemented in deriving class. This method has to return an integer. This represent the size of the BoardItem. It is used for example to evaluate the space taken in the inventory.

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.Characters.**Player**(***kwargs*)
Bases: *gamelib.Movable.Movable*, *gamelib.Characters.Character*

A class that represent a player controlled by a human. It accepts all the parameters from *Character* and is a *Movable*.

Note: If no inventory is passed as parameter a default one is created.

---

can_move()
Movable implements can_move().

Returns True

Return type Boolean

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

Return type str

display()
Print the model WITHOUT carriage return.

has_inventory()
This method returns True (a player has an inventory).

overlappable()
This method returns false (a player cannot be overlapped).

Note: If you wish your player to be overlappable, you need to inherit from that class and re-implement overlappable().

---

pickable()
This method returns False (a player is obviously not pickable).

size()
This is a virtual method that must be implemented in deriving class. This method has to return an integer. This represent the size of the BoardItem. It is used for example to evaluate the space taken in the inventory.

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

CHAPTER 4

Constants

Accessible constants are the following:

General purpose:

- HAC_GAME_LIB_VERSION

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
- NONE_AUTHORIZED

UI positions:

- POS_TOP
- POS_BOTTOM

- ORIENTATION_HORIZONTAL
- ORIENTATION_VERTICAL

Actions states (for Actuators for example):

- RUNNING
- PAUSED
- STOPPED

CHAPTER 5

Game

class gamelib.Game.**Game**(*name='Game'*, *boards={}*, *menu={}*, *current_level=None*)

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 *gamelib.Board.Board*), associate it to level, takes care of level changing, etc.

Parameters

- **name** (*str*) – The Game name.
- **boards** (*dict*) – A dictionnary of boards with the level number as key and a board reference as value.
- **menu** (*dict*) – A dictionnary of menus with a category (str) as key and another dictionnary (key: a shortcut, value: a description) as value.
- **current_level** (*int*) – The current level.

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.

actuate_npcs(*level_number*)

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.)

Parameters **level_number** – The number of the level to actuate NPCs in.

Example:

```
mygame.actuate_npcs(1)
```

Note: This method only move NPCs when their actuator state is RUNNING. If it is PAUSED or STOPPED, theNPC is not moved.

add_board(*level_number*, *board*)
Add a board for the level number.

This method associate a Board (*gamelib.Board.Board*) to a level number.

Example:

```
game.add_board(1,myboard)
```

Parameters

- **level_number** (*int*) – the level number to associate the board to.
- **board** (gamelib.Board.Board) – a Board object corresponding to the level number.

Raises ***HacInvalidTypeException*** – If either of these parameters are not of the correct type.

add_menu_entry(*category*, *shortcut*, *message*, *data=None*)
Add a new entry to the menu.

Add another shortcut and message to the specified category.

Categories help organize the different sections of a menu or dialogues.

Parameters

- **category** (*str*) – The category to which the entry should be added.
- **shortcut** (*str*) – A shortcut (usually one key) to display.
- **message** (*various*) – a message that explains what the shortcut does.
- **data** – a data that you can get from the menu object.

The shortcut and data is optional.

Example:

```
game.add_menu_entry('main_menu','d','Go right',Constants.RIGHT)
game.add_menu_entry('main_menu',None,'------------------')
game.add_menu_entry('main_menu','v','Change game speed')
```

add_npc(*level_number*, *npc*, *row=None*, *column=None*)
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** (gamelib.Characters.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 HacInvalidTypeException exception is raised.

Important: If the NPC does not have an actuator, this method is going to affect a gamelib.Actuators.SimpleActuators.RandomActuator() to npc.actuator. And if npc.step == None, this method sets it to 1

---

animate_items(*level_number*)

That method goes through all the BoardItems of a given map and call Animation.next_frame() :param level_number: The number of the level to animate items in. :type level_number: int

Raise *gamelib.HacExceptions.HacInvalidLevelException*
class:*gamelib.HacExceptions.HacInvalidTypeException*

Example:

```
mygame.animate_items(1)
```

change_level(*level_number*)

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 ***HacInvalidTypeException*** – If parameter is not an int.

clear_screen()

Clear the whole screen (i.e: remove everything written in terminal)

current_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 HacException exception is raised with an invalid_level error.

delete_menu_category(*category=None*)

Delete an entire category from the menu.

That function removes the entire list of messages that are attached to the category.

Parameters **category** (*str*) – The category to delete.

Raises ***HacInvalidTypeException*** – If the category is not a string

---

Important: If the entry have no shortcut it's advised not to try to update unless you have only one NoneType as a shortcut.

Example:

```
game.add_menu_entry('main_menu','d','Go right')
game.update_menu_entry('main_menu','d','Go LEFT',Constants.LEFT)
```

display_board()
Display the current board.

This is an alias for Game.current_board().display()

display_menu(*category*, *orientation=10010000*, *paginate=10*)
Display the menu.

This method display the whole menu for a given category.

Parameters

- **category** (*str*) – The category to display. **Mandatory** parameter.
- **orientation** (gamelib.Constants.Constants) – The shortcut of the entry you want to get.
- **paginate** (*int*) – pagination parameter (how many items to display before changing line or page).

Example:

```
game.display_menu('main_menu')
game.display_menu('main_menu', Constants.ORIENTATION_HORIZONTAL, 5)
```

display_player_stats(*life_model='\x1b[41m \x1b[0m'*, *void_model='\x1b[40m \x1b[0m'*)
Display the player name and health.

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_menu_entry(*category*, *shortcut*)
Get an entry of the menu.

This method return a dictionnary with 3 entries :

- shortcut
- message
- data

Parameters

- **category** (*str*) – The category in which the entry is located.
- **shortcut** (*str*) – The shortcut of the entry you want to get.

Returns The menu entry or None if none was found

Return type dict

Example:

```
ent = game.get_menu_entry('main_menu','d')
game.move_player(int(ent['data']),1)
```

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 *gamelib.Board.Board*)

Example:

```
mynewboard = game.load_board( 'awesome_level.json', 1 )
game.change_level( 1 )
```

load_config(*filename*, *section='main'*, *defaults={}*)
Load a configuration file from the disk. The configuration file must respect the INI syntax. The goal of these methods is to be 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.
- **defaults** (*dict*) – The default value for each variable in the config file (or not). If your config file uses sections, your defaults needs to represent that.

See https://docs.python.org/3/library/configparser.html for more information on that.

Example:

```
mygame.load_config('game_controls.ini','game_control')
```

move_player(*direction*, *step*)
Easy wrapper for Board.move().

Example:

```
mygame.move_player(Constants.RIGHT,1)
```

neighbors(*radius=1*, *object=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** (gamelib.BoardItem.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 ***HacInvalidTypeException*** – 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]})')
```

pause()

Set the game engine state to PAUSE.

Example:

```
mygame.pause()
```

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

- ***HacInvalidTypeException*** – If any parameter is not of the right type
- ***HacInvalidLevelException*** – 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.

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()
```

stop()

Set the game engine state to STOPPED.

Example:

```
mygame.stop()
```

update_menu_entry(*category*, *shortcut*, *message*, *data=None*)

Update an entry of the menu.

Update the message associated to a category and a shortcut.

Parameters

- **category** (*str*) – The category in which the entry is located.
- **shortcut** (*str*) – The shortcut of the entry you want to update.
- **message** (*various*) – a message that explains what the shortcut does.
- **data** – a data that you can get from the menu object.

Important: If the entry have no shortcut it's advised not to try to update unless you have only one NoneType as a shortcut.

Example:

```
game.add_menu_entry('main_menu','d','Go right')
game.update_menu_entry('main_menu','d','Go LEFT',Constants.LEFT)
```

CHAPTER 6

HacExceptions

This module regroup all the specific exceptions of the library. The idea behind most exceptions is to provide more context and info that the standard exceptions.

exception gamelib.HacExceptions.**HacException**(*error*, *message*)
Exception raised for non specific errors in HAC-GAME-LIB.

exception gamelib.HacExceptions.**HacInvalidLevelException**(*message*)
Exception raised if a level is not associated to a board in Game().

exception gamelib.HacExceptions.**HacInvalidTypeException**(*message*)
Exception raised for invalid types.

exception gamelib.HacExceptions.**HacInventoryException**(*error*, *message*)
Exception raised for issue related to the inventory. The error is an explicit string, and the message explains the error.

exception gamelib.HacExceptions.**HacObjectIsNotMovableException**(*message*)
Exception raised if the object that is being moved is not a subclass of Movable.

exception gamelib.HacExceptions.**HacOutOfBoardBoundException**(*message*)
Exception for out of the board's boundaries operations.

CHAPTER 7

Immovable

This module contains the Immovable and Actionable classes.

class gamelib.Immovable.**Actionable**(***kwargs*)

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

Parameters

- **action** (*function*) – the reference to a function (Attention: no parentheses at the end of the function name).
- **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*

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.

---

activate()

This function is calling the action function with the action_parameters.

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

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

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

Return type str

display()
Print the model WITHOUT carriage return.

overlappable()
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 BoardItem to be overlapped by another BoardItem.

pickable()
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 BoardItem to be pick-up by player or NPC.

restorable()
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 an Immovable BoardItem to be restored by the board if the item is overlappable and has been overlapped by another Movable (*Movable*) item.

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.Immovable.**Immovable**(***kwargs*)
This class derive BoardItem and describe an object that cannot move or be moved (like a wall). Thus this class implements BoardItem.can_move(). However it does not implement BoardItem.pickable() or BoardItem.overlappable()

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

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

Return type str

display()
Print the model WITHOUT carriage return.

overlappable()
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 BoardItem to be overlapped by another BoardItem.

pickable()
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 BoardItem to be pick-up by player or NPC.

restorable()
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 an Immovable BoardItem to be restored by the board if the item is overlappable and has been overlapped by another Movable (*Movable*) item.

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

CHAPTER 8

Inventory

This module contains the Inventory class.

class gamelib.Inventory.**Inventory**(*max_size=10*)

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.

The constructor takes only one parameter: the maximum size of the inventory. 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. Deafult value: 10.

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.

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.size is greater than the inventory max_size)

Parameters **item** (*BoardItem*) – the item you want to add

Raises HacInventoryException, HacInvalidTypeException

Example:

```
item = Treasure(model=Sprites.MONEY_BAG,size=2,name='Money bag')
try:
    mygame.player.inventory.add_item(item)
expect HacInventoryException 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)
```

Warning: if you try to add more than one item with the same name (or if the name is empty), this function will automatically change the name of the item by adding a UUID to it.

delete_item(*name*)

Delete the item corresponding to the name given in argument.

Parameters **name** (*str*) – the name of the item you want to delete.

Note: in case an execpetion is raised, the error will be 'no_item_by_that_name' and the message is giving the specifics.

See also:

gamelib.HacExceptions.HacInventoryException.

Example:

```
life_container = mygame.player.inventory.get_item('heart_1')
if isinstance(life_container,GenericActionableStructure):
    life_container.action(life_container.action_parameters)
    mygame.player.inventory.delete_item('heart_1')
```

get_item(*name*)

Return the 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*

Raises HacInventoryException

Note: in case an execpetion is raised, the error will be 'no_item_by_that_name' and the message is giving the specifics.

See also:

gamelib.HacExceptions.HacInventoryException.

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.

items_name()
Return the list of all items names in the inventory.

Returns a list of string representing the items names.

Return type list

search(*query*)
Search for objects in the inventory.

All objects that matches the query are going to be returned. :param query: the query that items in the inventory have to match to be returned :type name: str :returns: a table of BoardItems. :rtype: list

Example:

```
for item in game.player.inventory.search('mighty'):
    print(f"This is a mighty item: {item.name}")
```

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}")
```

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
```

CHAPTER 9

Movable

This module contains the Movable class. It can potentially hold more movement related classes.

class gamelib.Movable.**Movable**(***kwargs*)

A class representing BoardItem capable of movements.

Movable subclasses `BoardItem`.

Parameters **step** (*int*) – the amount of cell a movable can cross in one turn.

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()

This class contains a private member called _overlapping. This private member is used to store the reference to an overlappable object while a movable occupy its position. The Board then restore the overlapped object. You should let the Board class take care of that.

can_move()

Movable implements can_move().

Returns True

Return type Boolean

debug_info()

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

Return type str

display()

Print the model WITHOUT carriage return.

has_inventory()

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.

overlappable()

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 BoardItem to be overlapped by another BoardItem.

pickable()
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 BoardItem to be pick-up by player or NPC.

size()
This is a virtual method that must be implemented in deriving class. This method has to return an integer. This represent the size of the BoardItem. It is used for example to evaluate the space taken in the inventory.

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

CHAPTER 10

Sprites

Sprites are simply filtered emojis. Explore this file for a complete list. All emoji codes from: https://unicode.org/emoji/charts/full-emoji-list.html

The complete list of aliased emojis is:

- COWBOY =
- DEAMON_HAPPY =
- DAEMON_ANGRY =
- SKULL =
- SKULL_CROSSBONES =
- POO =
- CLOWN =
- OGRE =
- HAPPY_GHOST =
- ALIEN =
- ALIEN_MONSTER =
- ROBOT =
- CAT =
- CAT_FACE =
- CAT_LOVE =
- CAT_WEARY =
- CAT_CRY =
- CAT_ANGRY =
- HEART =

- HEART_SPARKLING =
- HEART_BROKEN =
- HEART_ORANGE =
- HEART_YELLOW =
- HEART_GREEN =
- HEART_BLUE =
- EXPLOSION =
- DIZZY =
- DASH =
- HOLE =
- BOMB =
- BRAIN =
- BOY =
- GIRL =
- MAN =
- MAN_BEARD =
- WOMAN =
- WOMAN_BLOND =
- MAN_OLD =
- WOMAN_OLD =
- POLICE =
- SUPER_HERO =
- SUPER_VILAIN =
- MAGE =
- FAIRY =
- VAMPIRE =
- MERMAID =
- ELF =
- GENIE =
- ZOMBIE =
- PERSON_RUNNING =
- PERSON_WALKING =
- PERSON_FENCING =
- PERSON_SLEEPING =
- PERSON_YOGA =
- PERSON_BATHING =

- MONKEY =
- GORILLA =
- DOG =
- DOG_FACE =
- WOLF_FACE =
- FOX_FACE =
- RACCOON_FACE =
- LION_FACE =
- TIGER_FACE =
- HORSE_FACE =
- HORSE =
- UNICORN_FACE =
- DEER_FACE =
- COW_FACE =
- COW =
- OX =
- BUFFALO =
- PIG =
- PIG_FACE =
- RAM =
- SHEEP =
- GOAT =
- LLAMA =
- GIRAFFE =
- ELEPHANT =
- RHINOCEROS_FACE =
- MOUSE =
- RABBIT =
- CHIPMUNK =
- BAT =
- PANDA_FACE =
- TURKEY =
- CHICKEN =
- CHICK =
- EAGLE =
- DUCK =

- OWL =
- FROG_FACE =
- CROCODILE =
- TURTLE =
- LIZARD =
- SNAKE =
- DRAGON =
- DINOSAUR =
- TREX =
- WHALE =
- DOLPHIN =
- SHARK =
- OCTOPUS =
- SPIDER =
- SPIDER_WEB =
- SCORPION =
- MICROBE =
- SUNFLOWER =
- CHERRY_BLOSSOM =
- FLOWER =
- ROSE =
- TREE_PINE =
- TREE =
- TREE_PALM =
- CACTUS =
- CLOVER =
- CLOVER_LUCKY =
- CHEESE =
- MEAT_BONE =
- MEAT =
- BACON =
- EGG =
- CRAB =
- LOBSTER =
- SHRIMP =
- SQUID =

- KNIFE =
- AMPHORA =
- EARTH_GLOBE =
- WALL =
- HOUSE =
- CASTLE =
- MON =
- FOUNTAIN =
- ROCKET =
- FLYING_SAUCER =
- HOURGLASS =
- CYCLONE =
- RAINBOW =
- ZAP =
- SNOWMAN =
- COMET =
- FIRE =
- WATER_DROP =
- JACK_O_LANTERN =
- DYNAMITE =
- SPARKLES =
- GIFT =
- TROPHY =
- CROWN =
- GEM_STONE =
- CANDLE =
- LIGHT_BULB =
- BOOK_OPEN =
- SCROLL =
- MONEY_BAG =
- BANKNOTE_DOLLARS =
- BANKNOTE_EUROS =
- BANKNOTE_WINGS =
- DOLLAR =
- LOCKED =
- UNLOCKED =

- KEY =
- PICK =
- SWORD =
- SWORD_CROSSED =
- PISTOL =
- BOW =
- SHIELD =
- COFFIN =
- RADIOACTIVE =
- FLAG_GOAL =
- DOOR =

CHAPTER 11

Structures

This module contains many "helpers" classes to populate your game with structures. It contains many directly usable structures and some generic ones that can be turned in anything you like.

class gamelib.Structures.**Door**(***kwargs*)

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.
- **size** (*str*) – The size of the door. 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=Sprites.DOOR,type='locked_door')
```

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

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

Return type str

display()
Print the model WITHOUT carriage return.

overlappable()
This represent the capacity for a *BoardItem* to be overlapped by player or NPC.

To set this value please use *set_overlappable()*

Returns False

Return type bool

See also:

set_overlappable()

pickable()
This represent the capacity for a BoardItem to be picked-up by player or NPC.

To set this value please use *set_pickable()*

Returns True or False

Return type bool

See also:

set_pickable()

restorable()
This represent the capacity for an *Immovable BoardItem* (in this case a GenericStructure item) to be restored by the board if the item is overlappable and has been overlapped by another *Movable* item.

The value of this property is set with *set_restorable()*

Returns False

Return type bool

See also:

set_restorable()

set_overlappable(*val*)
Make the structure overlappable or not.

Parameters **val** (*bool*) – True or False depending on the fact that the structure can be overlapped (i.e that a Player or NPC can step on it) or not.

Example:

```
myneatstructure.set_overlappable(True)
```

set_pickable(*val*)
Make the structure pickable or not.

Parameters **val** (*bool*) – True or False depending on the pickability of the structure.

Example:

```
myneatstructure.set_pickable(True)
```

set_restorable(*val*)
Make the structure restorable or not.

Parameters **val** (*bool*) – True or False depending on the restorability of the structure.

Example:

```
myneatstructure.set_restorable(True)
```

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.Structures.**GenericActionableStructure**(***kwargs*)
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.

activate()
This function is calling the action function with the action_parameters.

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

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

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

Return type str

display()
Print the model WITHOUT carriage return.

overlappable()
This represent the capacity for a *BoardItem* to be overlapped by player or NPC.

To set this value please use *set_overlappable()*

Returns False

Return type bool

See also:

set_overlappable()

pickable()
This represent the capacity for a BoardItem to be picked-up by player or NPC.

To set this value please use *set_pickable()*

Returns True or False

Return type bool

See also:

set_pickable()

restorable()
This represent the capacity for an *Immovable BoardItem* (in this case a GenericStructure item) to be restored by the board if the item is overlappable and has been overlapped by another *Movable* item.

The value of this property is set with *set_restorable()*

Returns False

Return type bool

See also:

set_restorable()

set_overlappable(*val*)
Make the structure overlappable or not.

Parameters **val** (*bool*) – True or False depending on the fact that the structure can be overlapped (i.e that a Player or NPC can step on it) or not.

Example:

```
myneatstructure.set_overlappable(True)
```

set_pickable(*val*)
Make the structure pickable or not.

Parameters **val** (*bool*) – True or False depending on the pickability of the structure.

Example:

```
myneatstructure.set_pickable(True)
```

set_restorable(*val*)
Make the structure restorable or not.

Parameters **val** (*bool*) – True or False depending on the restorability of the structure.

Example:

```
myneatstructure.set_restorable(True)
```

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.Structures.**GenericStructure**(***kwargs*)
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 *gamelib.Structures.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.

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.

---

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

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

Return type str

display()
Print the model WITHOUT carriage return.

overlappable()
This represent the capacity for a *BoardItem* to be overlapped by player or NPC.

To set this value please use *set_overlappable()*

Returns False

Return type bool

See also:

set_overlappable()

pickable()
This represent the capacity for a BoardItem to be picked-up by player or NPC.

To set this value please use *set_pickable()*

Returns True or False

Return type bool

See also:

set_pickable()

restorable()
This represent the capacity for an *Immovable BoardItem* (in this case a GenericStructure item) to be restored by the board if the item is overlappable and has been overlapped by another *Movable* item.

The value of this property is set with *set_restorable()*

Returns False

Return type bool

See also:

set_restorable()

set_overlappable(*val*)
Make the structure overlappable or not.

Parameters **val** (*bool*) – True or False depending on the fact that the structure can be overlapped (i.e that a Player or NPC can step on it) or not.

Example:

```
myneatstructure.set_overlappable(True)
```

set_pickable(*val*)
Make the structure pickable or not.

Parameters **val** (*bool*) – True or False depending on the pickability of the structure.

Example:

```
myneatstructure.set_pickable(True)
```

set_restorable(*val*)
Make the structure restorable or not.

Parameters **val** (*bool*) – True or False depending on the restorability of the structure.

Example:

```
myneatstructure.set_restorable(True)
```

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.Structures.**Treasure**(***kwargs*)
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.
- **size** (*str*) – The size of 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=Sprites.MONEY_BAG,value=100,size=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}")
```

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

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

Return type str

display()
Print the model WITHOUT carriage return.

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

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

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

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)
Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

class gamelib.Structures.**Wall**(***kwargs*)
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.

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

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

Return type str

display()
Print the model WITHOUT carriage return.

overlappable()
This represent the capacity for a *BoardItem* to be overlapped by player or NPC.

Returns False

Return type bool

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?')
```

restorable()
This represent the capacity for an *Immovable* Movable item. A wall is not overlappable.

Returns False

Return type bool

size()
Return the size of the Immovable Item.

Returns The size of the item.

Return type int

store_position(*row*, *column*)

Store the BoardItem position for self access.

The stored position is used for consistency and quick access to the self postion. 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*.

Example:

```
item.store_position(3,4)
```

CHAPTER 12

Utils

This module regroup different utility functions and constants.

gamelib.Utils.**black**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**black_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**black_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**blue**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**blue_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**blue_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**clear_screen**()
This methods clear the screen

gamelib.Utils.**cyan**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**cyan_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**cyan_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**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:

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

gamelib.Utils.**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:

```
Utils.fatal("|x_x|")
```

gamelib.Utils.**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.

gamelib.Utils.**green**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**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( Utils.green_bright("This is a formatted message") )
```

gamelib.Utils.**green_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**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:

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

gamelib.Utils.**init_term_colors**()
This function is a forward to colorama.init()

gamelib.Utils.**magenta**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**magenta_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**magenta_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**print_white_on_red**(*message*)
Print a white message over a red background.

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

Example:

```
Utils.print_white_on_red("This is bright!")
```

gamelib.Utils.**red**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**red_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**red_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**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:

```
Utils.warn("This is a warning.")
```

gamelib.Utils.**white**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**white_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**white_dim**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**yellow**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**yellow_bright**(*message*)
This method works exactly the way green_bright() work with different color.

gamelib.Utils.**yellow_dim**(*message*)
This method works exactly the way green_bright() work with different color.

CHAPTER 13

Actuators

13.1 SimpleActuators

This module contains the simple actuators classes. Simple actuators are movement related one. They allow for predetermined movements patterns.

class gamelib.Actuators.SimpleActuators.**PathActuator**(*path=None*)

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.

next_move()

Return the movement based on current index

The movement is selected from path if state is RUNNING, otherwise it should return None. 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 | None

Example:

```
pathactuator.next_move()
```

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

set_path(*path*)
Defines a new path

This will also reset the index back to 0.

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

Example:

```
pathactuator.set_path([Constants.UP,Constants.DOWN,Constants.LEFT,Constants.
↪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()
```

class gamelib.Actuators.SimpleActuators.**PatrolActuator**(*path=None*)
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 beggining of the list. Once it reaches the beggining 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 paths.

next_move()
Return the movement based on current index

The movement is selected from path if state is RUNNING, otherwise it should return None. 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 | None

Example:

```
patrolactuator.next_move()
```

pause()
Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

set_path(*path*)
Defines a new path

This will also reset the index back to 0.

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

Example:

```
pathactuator.set_path([Constants.UP,Constants.DOWN,Constants.LEFT,Constants.
↪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()
```

class gamelib.Actuators.SimpleActuators.**RandomActuator**(*moveset=None*)
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.

next_move()
Return a randomly selected movement

The movement is randomly selected from moveset if state is RUNNING, otherwise it should return None.

Returns The next movement

Return type int | None

Example:

```
randomactuator.next_move()
```

pause()
Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

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()
```

13.2 AdvancedActuators

This module contains the more advanced actuators. AdvancedActuators allow for more actions and not only movement. It can also be more advanced movement classes.

class gamelib.Actuators.AdvancedActuators.**PathFinder**(*game=None*, *actuated_object=None*, *circle_waypoints=True*)

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** (gamelib.Game.Game) – A reference to the instanciated game engine.
- **actuated_object** (gamelib.BoardItem.BoardItem) – The 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)

add_waypoint(*row*, *column*)
Add a waypoint to the list of waypoints.

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

Parameters

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

Raises ***HacInvalidTypeException*** – 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)
```

clear_waypoints()
Empty the waypoints stack.

Example:

```
pf.clear_waypoints()
```

current_path()
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()
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)
```

find_path()
Find a path to the destination.

Destination (PathFinder.destination) has to be set beforehand. This method implements a Breadth First Search algorithm (Wikipedia) to find the shortest path to destination.

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.

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

Raises NotImplementedError

next_move()
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 betwen 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=Sprites.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 == Constants.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()
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)
```

pause()
Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

remove_waypoint(*row*, *column*)
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 HacInvalidTypeException is raised.

Parameters

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

Raises

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

Example:

```
method()
```

set_destination(*row=0*, *column=0*)
Set the targeted destination.

Parameters

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

Raises ***HacInvalidTypeException*** – 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()
```

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 inheritate from one of these base class.

class gamelib.Actuators.Actuator.**Actuator**

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.

next_move()

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

Raises NotImplementedError

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

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()
```

class gamelib.Actuators.Actuator.**Behavioral**

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.

next_action()

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

Raises NotImplementedError

next_move()

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

Raises NotImplementedError

pause()

Set the actuator state to PAUSED.

Example:

```
mygame.pause()
```

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()
```

CHAPTER 14

Animation

This module contains the animation relation classes (so far only Animation).

class gamelib.Animation.**Animation**(*display_time=0.05*, *auto_replay=True*, *frames=None*, *animated_object=None*, *refresh_screen=None*)

The Animation class is used to give the ability to have more than one model for a BoardItem. An 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]*) – an array of "frames" (string)
- **animated_object** (*BoardItem*) – The object to animate.
- **refresh_screen** (*function*) – The callback function that controls the redrawing of the screen. This function reference should come from the main game.

Example

```
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
# explicitely create one
item.animation = Animation(display_time=0.1, animated_object=item,
                           refresh_screen=redraw_screen)
```

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*) – The frame to add to the animation.

Raise *gamelib.HacExceptions.HacInvalidTypeException*

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()
```

next_frame()
Update the animated_object.model with the next frame of the animation.

That method takes care of automatically replaying the animation if the last frame is reached if the state is RUNNING.

If the the state is PAUSED it still update the animated_object.model and returning the current frame. It does NOT actually go to next frame.

If animated_object is not a sub class of *BoardItem* an exception is raised.

Raise *HacInvalidTypeException*

Example:

```
item.animation.next_frame()
```

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 animated_object 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 *HacInvalidTypeException*

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, HacInvalidTypeException

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 *gamelib.HacExceptions.HacInvalidTypeException*

Example:

```
item.animation.remove_frame(
    item.animation.search_frame(Sprite.ALIEN_MONSTER)
)
```

start()
Set the animation state to RUNNING.

If the animation state is not 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()
```

CHAPTER 15

Credits

15.1 Development Leads

- Arnaud Dupuis (@arnauddupuis)
- Kalil de Lima (@kaozdl)

15.2 Contributors

- 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)

CHAPTER 16

History

16.1 1.0.1 (2020-05-17)

- Fix a huge default save directory issue (see bellow) 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).

16.2 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

16.3 2019.5

- Please see the official website.

16.4 pre-2019.5

- Please see the Github for history.

CHAPTER 17

Forewords

This python3 module is a base for the programming lessons of the Hyrule Astronomy Club. It is not meant to be a comprehensive game building library.

It is however meant (and used) to teach core programming concept to kids from age 6 to 13.

CHAPTER 18

Introduction

First of all, his module is exclusively compatible with python 3.

The core concept is that it revolve around the *Game* object, the *Board* object and the derivatives of *BoardItem*.

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

The base game makes use of:

- The main "game engine" (gamelib.Game.Game)
- **Many different types of structures (from gamelib.Structures), like:**
  - Wall (well the walls. . .),
  - Treasure (gems and money bag),
  - GenericStructure (trees),
  - GenericActionnableStructure (hearts and portals).
- Game()'s menu capabilities.
- Player and NPC (from gamelib.Characters)
- Inventory (from gamelib.Inventory)
- Player and Inventory stats
- **Simple actuators (gamelib.SimpleActuators) like:**
  - RandomActuator (NPCs in level 2),
  - PathActuator (NPCs in level 1).

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Y