Garden Gnome
Assignment 2
Semester 2, 2022
CSSE1001/CSSE7030

v1.3.1 - 15th Sept

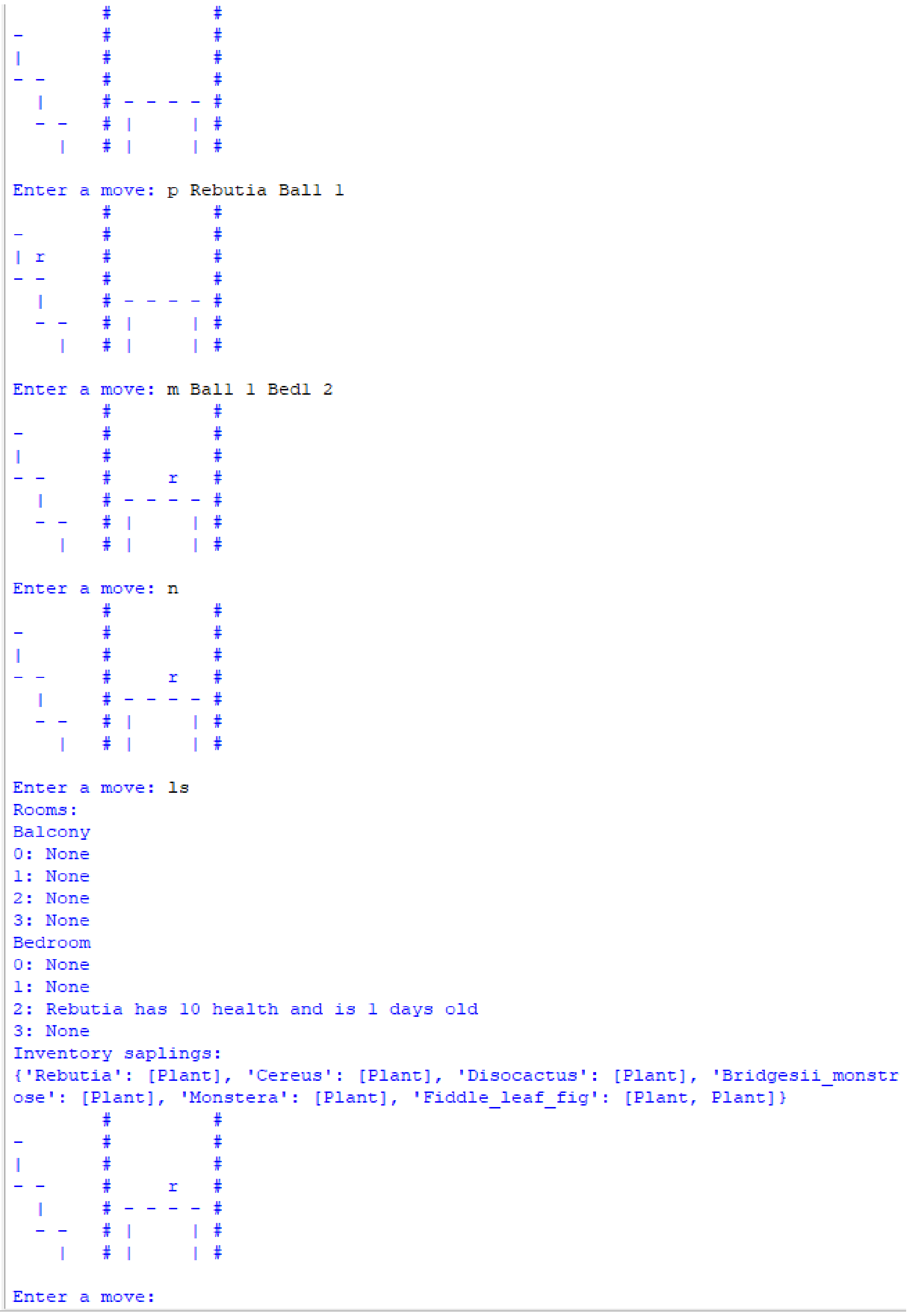
Due date: September 21, 2022 20:00 AEST

1 Introduction

Garden Gnome is a single-player simulator game where a player must take care of house plants for 15 days. The player must keep more than half their plants alive for this duration. Items are be rewarded throughout the days to assist the player, and are applied when needed. See Figure 1 for a sample game.

In Assignment 2, you will create an extensible object-oriented implementation of *Garden Gnome* which employs a text-based interface.

You are required to implement a collection of classes and methods as specified in Section 5 of this document. Your program's output must match the expected output *exactly*; minor differences in output (such as white-space or casing) *will* cause tests to fail, resulting in *zero marks* for that test. Any changes to this document will be listed in a changelog at the end of the document.



```
        #         #
-       #         #
|       #         #
- -     #         #
  |     # - - - - #
  - -   # |     | #
    |   # |     | #

Enter a move: p Rebutia Ball 1
        #         #
-       #         #
| r     #         #
- -     #         #
  |     # - - - - #
  - -   # |     | #
    |   # |     | #

Enter a move: m Ball 1 Bed1 2
        #         #
-       #         #
|       #         #
- -     #     r   #
  |     # - - - - #
  - -   # |     | #
    |   # |     | #

Enter a move: n
        #         #
-       #         #
|       #         #
- -     #     r   #
  |     # - - - - #
  - -   # |     | #
    |   # |     | #

Enter a move: ls
Rooms:
Balcony
0: None
1: None
2: None
3: None
Bedroom
0: None
1: None
2: Rebutia has 10 health and is 1 days old
3: None
Inventory saplings:
{'Rebutia': [Plant], 'Cereus': [Plant], 'Disocactus': [Plant], 'Bridgesii_monstr
ose': [Plant], 'Monstera': [Plant], 'Fiddle_leaf_fig': [Plant, Plant]}
        #         #
-       #         #
|       #         #
- -     #     r   #
  |     # - - - - #
  - -   # |     | #
    |   # |     | #

Enter a move:
```

Figure 1: A snippet from a functional game of Garden Gnome. Each room in the house is separated by walls (represented by #) and each room can have four plants eac represented by a single character.

2 Getting Started

Download `a2.zip` from Blackboard — this archive contains the necessary files to start this assignment.

Once extracted, the `a2.zip` archive will provide the following files/directories:

`a2.py`
: The game engine. This is *the only file you submit and modify. Do not* make changes to any other files.

`a2_support.py`
: Provided classes implementing functionality to render the text interface.

`constants.py`
: Constants you *may* find helpful when writing your assignment.

`houses`
: A folder containing several text files of playable houses.

`example_simulations`
: A folder containing example output from running the completed assignment.

3 Gameplay

The objective is to have more than half the plants alive at day 15. For example, if the game started with ten plants, *six* plants are required to be alive at day 15 to win the game.

Gameplay occurs according to the following steps:

1. Prompt user with "`Enter house file: `". Presume the file exists at the location given relative to where `a2.py` is saved. For instance, `houses/house1.txt`.
2. Display house and prompt user with "`Enter a move: `". The list of valid moves are given below. Assume invalid moves may be entered, but ignore them.
3. Progress to next day when `n` is entered.
4. Break on lose or win, otherwise go to step 2.

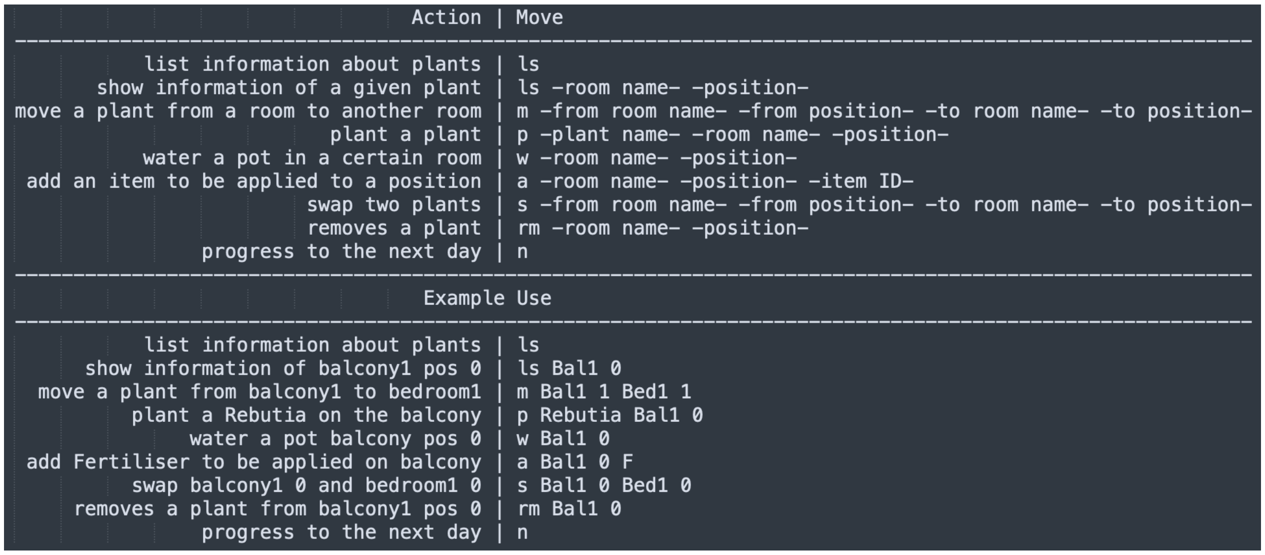

```
                                  Action | Move
--------------------------------------------------------------------------------------------------------------
          list information about plants | ls
      show information of a given plant | ls -room name- -position-
move a plant from a room to another room | m -from room name- -from position- -to room name- -to position-
                          plant a plant | p -plant name- -room name- -position-
          water a pot in a certain room | w -room name- -position-
 add an item to be applied to a position | a -room name- -position- -item ID-
                        swap two plants | s -from room name- -from position- -to room name- -to position-
                        removes a plant | rm -room name- -position-
               progress to the next day | n
--------------------------------------------------------------------------------------------------------------
                                  Example Use
--------------------------------------------------------------------------------------------------------------
          list information about plants | ls
     show information of balcony1 pos 0 | ls Bal1 0
  move a plant from balcony1 to bedroom1 | m Bal1 1 Bed1 1
         plant a Rebutia on the balcony | p Rebutia Bal1 0
              water a pot balcony pos 0 | w Bal1 0
 add Fertiliser to be applied on balcony | a Bal1 0 F
         swap balcony1 0 and bedroom1 0 | s Bal1 0 Bed1 0
    removes a plant from balcony1 pos 0 | rm Bal1 0
               progress to the next day | n
```

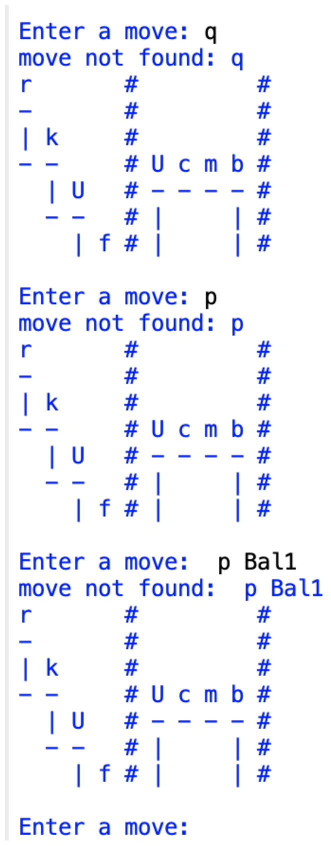
Figure 2: All available moves.

3.1 Room and position system

The room and position system used for the simulator is depicted in Figure 4. For example in the "Balcony" room there are always four possible positions to place plants, 0, 1, 2, and 3, from left to right, each room has 4 columns at different elevations. Positions are represented as a single `int`. Note that this is just an example; you must *not* assume that you will always have a Balcony room.

4 Class overview and relationships

- Hollow-arrowheads indicate *inheritance* (i.e. the "is-a" relationship).
- Dotted arrows indicates *composition* (i.e. the "has-a" relationship). An arrow marked with $\xrightarrow{1\ 1}$ denotes that each instance of the class at the base of the arrow contains exactly one instance of the class at the head of the arrow. An arrow marked with $\xrightarrow{1\ n}$ denotes that each instance of the class at the base of the arrow may contain many instances of the class at the head of the arrow. E.g. a `Model` instance may contain many different `Room` instances, but only one `Inventory` instance.
- Green classes are *abstract* classes. You should only ever instantiate the blue



```
Enter a move: q
move not found: q
r       #         #
-       #         #
| k     #         #
- -     # U c m b #
  | U   # - - - - #
  - -   # |     | #
    | f # |     | #

Enter a move: p
move not found: p
r       #         #
-       #         #
| k     #         #
- -     # U c m b #
  | U   # - - - - #
  - -   # |     | #
    | f # |     | #

Enter a move:  p Bal1
move not found:  p Bal1
r       #         #
-       #         #
| k     #         #
- -     # U c m b #
  | U   # - - - - #
  - -   # |     | #
    | f # |     | #

Enter a move:
```

Figure 3: Example behaviour when user input is invalid.

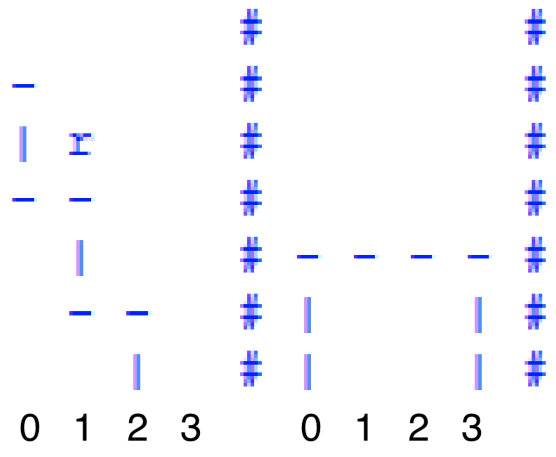


Figure 4: Room and position system for a Balcony

classes in your program, though you should instantiate the green classes to test them before beginning work on their subclasses.

- The class surrounded by a pink box is provided to you in `a2_support.py`.

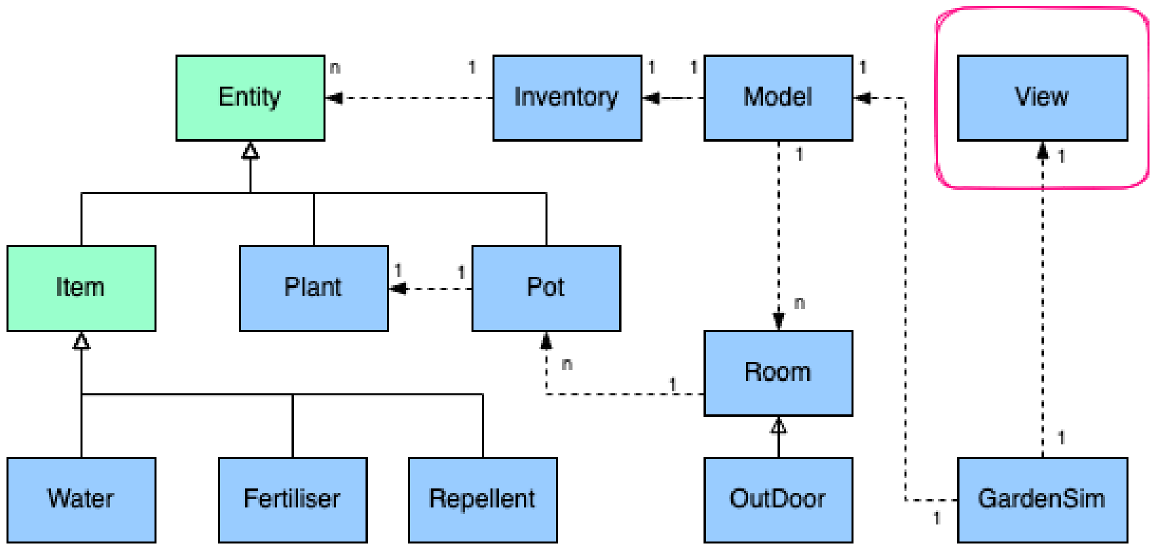


Figure 5: Basic class relationship diagram for the classes which need to be implemented for this assignment.

Note that all view classes have been provided to you. You are only required to implement a number of modelling classes and a single controller class which unites these modelling classes with the existing `View` view class.

5 Implementation

5.1 Model classes

This section outlines the model classes you must implement, along with their required methods. The classes are laid out in the order in which it is recommended you attempt them (with some exceptions that are discussed in the relevant sections). It is ***strongly recommended*** that you ensure a class is working via testing the implementation yourself before moving on to another class (particularly if that class is a superclass of the next class you will attempt).

Entity *(abstract class)*

Abstract base class for any entity. Provides base functionality for all entities in the game.

`get_class_name(self) -> str` *(method)*

Return the class name of this entity's class.

`get_id(self) -> str` *(method)*

Return the single character id of this entity's class. See `constants.py` for the ID value of all tiles and entities.

`__str__(self) -> str` *(method)*

Return the string representation for this Entity.

`__repr__(self) -> str` *(method)*

Return the text that would be required to make a new instance of this class that looks identical (where possible) to `self`.

Note: you are only required to replicate information that can be set via arguments to the `__init__` method for the relevant class. For instance, in the `Plant` subclass of `Entity` described later in this document, you are not required to provide the syntax for setting the new `Plant` instance's water, health, age, and repellent. Only the name needs to be communicated in the `__repr__` output, because this is the only attribute that can be replicated via the `Plant.__init__` method.

```
>>> entity = Entity()
>>> entity.get_class_name()
'Entity'
>>> entity.get_id()
'E'
>>> str(entity)
'E'
>>> entity
Entity()
```

Plant *(class)*

Inherits from Entity
Plant is an Entity that is planted by the user. A plant has water and health points

(HP) which start at 10, age starting at 0, and no repellent. A plant's drink rate and sun level are determined by the name of the plant See `constants.py` for the drink rate and sun level of all plants.

`__init__(self, name: str) -> None` *(method)*

Setup the plant with a given plant name. See `constants.py` for the plant names.

`get_name(self) -> str` *(method)*

Return name of the plant.

`get_drink_rate(self) -> float` *(method)*

Return water drinking rate of the plant

`decrease_water(self, amount: float) -> None` *(method)*

Decrease the plants water by a specified amount.

`drink_water(self) -> None` *(method)*

Reduce water levels by plant's drink rate. If water levels is zero (or less than zero) the plant's HP reduces by 1.

`get_sun_levels(self) -> tuple[int, int]` *(method)*

Return the acceptable sun level of the plant with the lower and upper range.

`get_health(self) -> int` *(method)*

Return the plant's current HP.

`add_health(self, amount: int) -> None` *(method)*

Add to the plant's health levels by a specified amount.

`decrease_health(self, amount: int = 1) -> None` *(method)*

Decrease the plants health by a specified amount, decrease by 1 by default.

`get_water(self) -> float` *(method)*

Return the water levels of the plant.

`water_plant(self) -> None` *(method)*

Add to the plant's water level by 1.

`set_repellent(self, applied: bool) -> None` *(method)*

Apply or remove repellent from plant.

`has_repellent(self) -> bool` *(method)*

Return `True` if the plant has repellent, `False` otherwise.

`get_age(self) -> int` *(method)*

Return how many days this plant has been planted.

`increase_age(self) -> None` *(method)*

Increase the number of days this plant has been planted by 1.

`is_dead(self) -> bool` *(method)*

Return `True` if the plant's health is less than or equals to zero, `False` otherwise.

```
>>> plant = Plant('Rebutia')
>>> plant.get_name()
'Rebutia'
>>> plant.get_health()
10
>>> plant.get_water()
10.0
>>> plant.get_drink_rate()
0.1
>>> plant.decrease_water(0.2)
>>> plant.get_water()
9.8
>>> plant.drink_water()
>>> plant.get_water()
9.700000000000001
>>> plant.water_plant()
>>> plant.get_water()
10.700000000000001
>>> plant.add_health(1)
>>> plant.get_health()
11
>>> plant.add_health(4)
>>> plant.get_health()
15
```

```
>>> plant.decrease_health()
>>> plant.get_health()
14
>>> plant.decrease_health(5)
>>> plant.get_health()
9
>>> plant.has_repellent()
False
>>> plant.set_repellent(True)
>>> plant.has_repellent()
True
>>> plant.get_age()
0
>>> plant.increase_age()
>>> plant.get_age()
1
>>> plant.decrease_water(11)
>>> plant.get_health()
9
>>> plant.get_water()
-0.29999999999999893
>>> plant.drink_water()
>>> plant.get_health()
8
>>> plant.is_dead()
False
>>> plant.decrease_health(8)
>>> plant.get_health()
0
>>> plant.is_dead()
True
>>> plant.get_class_name()
'Plant'
>>> plant.get_id()
'P'
>>> str(plant)
'P'
```

```
>>> plant
Plant('Rebutia')
```

Item *(abstract class)*

Inherits from Entity
Abstract subclass of Entity which provides base functionality for all items in the game.

apply(self, plant: Plant) -> None *(abstract method)*

Applies the items effect, if any, to the given plant. Raise NotImplementedError.

```
>>> plant = Plant('Rebutia')
>>> item = Item()
>>> item.apply(plant)
Traceback (most recent call last):
...
NotImplementedError
```

Water *(class)*

Inherits from Item
Adds to plant's water level by 1 when applied.

```
>>> plant = Plant('Rebutia')
>>> water = Water()
>>> plant.get_water()
10.0
>>> water.apply(plant)
>>> plant.get_water()
11.0
```

Fertiliser *(class)*

Inherits from Item

Adds to plant's health by 1 when applied.

```
>>> plant = Plant('Rebutia')
>>> fertiliser = Fertiliser()
>>> plant.get_health()
10
>>> fertiliser.apply(plant)
>>> plant.get_health()
11
```

PossumRepellent *(class)*

Inherits from Item
Cancel a possum attach when applied.

```
>>> plant = Plant('Rebutia')
>>> repellent = PossumRepellent()
>>> plant.has_repellent()
False
>>> repellent.apply(plant)
>>> plant.has_repellent()
True
```

Inventory *(class)*

An Inventory contains and manages a collection of items and plant.

`__init__(self, initial_items: Optional[list[Item]] = None, initial_plants: Optional[list[Plant]] = None) -> None` *(method)*

Sets up initial inventory. If no `initial_items` or `initial_plants` are provided, inventory starts with an empty dictionary for the entities. Otherwise, the initial dictionary is set up from the `initial_items` and `initial_plants` lists to be a dictionary mapping entity names to a list of entity instances with that name. **Note:** Plant is a plant object at age 0.

`add_entity(self, entity: Item | Plant) -> None` *(method)*

Adds the given item or plant to this inventory's collection of entities.

`get_entities(self, entity_type: str) -> dict[str, list[Item | Plant]]` *(method)*

Returns the a dictionary mapping entity (item or plant) names to the instances of the entity with that name in the inventory, respectively. **Note:** entity_type: The type can either be 'Plant' or 'Item'.

`remove_entity(self, entity_name: str) -> Optional[Item | Plant]` *(method)*

Removes one instance of the entity (item or plant) with the given name from inventory,if one exists. If no entity exists in the inventory with the given name, then this method returns None.

`__str__(self) -> str` *(method)*

Returns a string containing information about quantities of items available in the inventory.

`__repr__(self) -> str` *(method)*

Returns a string that could be used to construct a new instance of `Inventory` containing the same items as `self` currently contains. Note that the order of the `initial_items` is not important for this method.

```
>>> items = [Water(), Fertiliser()]
>>> plants = [Plant('Rebutia'), Plant('Monstera'), Plant('Rebutia')]
>>> inventory = Inventory(items, plants)
>>> inventory
Inventory(initial_items=[Water(), Fertiliser()], initial_plants=
[Plant('Rebutia'), Plant('Rebutia'), Plant('Monstera')])
>>> print(str(inventory))
W: 1
F: 1
Rebutia: 2
Monstera: 1
>>> inventory.get_entities('Item')
{'W': [Water()], 'F': [Fertiliser()]}
>>> inventory.add_entity(Plant('FiddleLeafFig'))
>>> inventory.get_entities('Plant')
```

```
{'Rebutia': [Plant('Rebutia'), Plant('Rebutia')], 'Monstera':
[Plant('Monstera')], 'FiddleLeafFig': [Plant('FiddleLeafFig')]}
>>> inventory.remove_entity('Rebutia')
Plant('Rebutia')
>>> inventory.get_entities('Plant')
{'Rebutia': [Plant('Rebutia')], 'Monstera': [Plant('Monstera')],
'FiddleLeafFig': [Plant('FiddleLeafFig')]}
>>> inventory.remove_entity(WATER)
Water()
>>> inventory.get_entities('Item')
{'F': [Fertiliser()]}
>>> print(str(inventory))
F: 1
Rebutia: 1
Monstera: 1
FiddleLeafFig: 1
```

Pot *(class)*

Pot is an Entity that has growing conditions information and an instance of plant.

`__init__(self) -> None` *(method)*

Sets up an empty pot.

`set_sun_range(self, sun_range: tuple[int, int]) -> None` *(method)*

Sets the sun range experienced by the pot.

`get_sun_range(self) -> tuple[int, int]` *(method)*

Returns the sun range experienced by the pot.

`set_evaporation(self, evaporation: float) -> None` *(method)*

Sets the evaporation rate of the pot.

`get_evaporation(self) -> float` *(method)*

Returns the evaporation rate of the pot.

`put_plant(self, plant: Plant) -> None` *(method)*

Adds an instance of a plant to the pot.

`look_at_plant(self) -> Optional[Plant]` *(method)*

Returns the plant in the pot and without removing it.

`remove_plant(self) -> Optional[Plant]` *(method)*

Returns the plant in the pot and removes it from the pot.

`progress(self) -> None` *(method)*

Progress the state of the plant and check if the current plant is suitable in the given conditions. Decrease the plant's water levels based on the evaporation. The health of the plant should decrease by 1:

- If the sun is not in a suitable range
- If the plant's water levels is below zero.

`animal_attack(self) -> None` *(method)*

Decreases the health of the plant by the animal attack damage dealt if a plant is in the pot and print out: `There has been an animal attack!` Do nothing if there is no plant in the pot. Attack should fail if plant has animal repellent. It should not affect the plant's health and print out: `There has been an animal attack! But luckily the {plant_name} has repellent.`

```
>>> plant = Plant('Rebutia')
>>> pot = Pot()
>>> print(str(pot))
Pot
>>> pot
Pot()
>>> pot.look_at_plant()
>>> pot.put_plant(plant)
>>> pot.look_at_plant()
Plant('Rebutia')
>>> pot.get_sun_range()
>>> pot.set_sun_range((5, 8))
>>> pot.get_sun_range()
(5, 8)
>>> pot.get_evaporation()
>>> pot.set_evaporation(1.1)
>>> pot.get_evaporation()
1.1
>>> plant = pot.remove_plant()
>>> pot.look_at_plant()
>>> plant
Plant('Rebutia')
>>> pot.put_plant(plant)
>>> pot.progress()
>>> pot.look_at_plant()
Plant('Rebutia')
>>> pot.look_at_plant().get_sun_levels()
(6, 10)
>>> pot.set_sun_range((4, 5))
>>> pot.progress()
Poor Rebutia dislikes the sun levels.
>>> pot.animal_attack()
>>> There has been an animal attack! Poor Rebutia.
>>> pot.look_at_plant().get_health()
4
>>> pot.look_at_plant().set_repellent(True)
>>> pot.animal_attack()
>>> There has been an animal attack! But luckily the Rebutia has
repellent.
>>> pot.look_at_plant().decrease_health(4)
>>> pot.progress()
Rebutia is dead
```

Room *(class)*

A Room instance represents the space in which plants can be planted and the instances of plants within the room.

`__init__(self, name: str) -> None` *(method)*

Set up an empty room of given room name. **Note:** Make use of *constants.py*.

`get_plants(self) -> dict[int, Plant | None]` *(method)*

Return the Plant instances in this room. with the keys being the positions and value being the corresponding plant, None if no plant is in the position.

`get_number_of_plants(self) -> int` *(method)*

Return the total number of live plants in the room.

`add_pots(self, pot_data: dict[int, Pot]) -> None` *(method)*

Add a pots to the room. Each key corresponds to a position in the room, with each value being an instance of a pot.

`get_pots(self) -> dict[int, Pot]` *(method)*

Return all pots within the room.

`get_pot(self, position: int) -> Pot` *(method)*

Return the Pot instance at the given position.

`add_plant(self, position: int, plant: Plant) -> None` *(method)*

Add a plant instance to Pot at a given position if no plant exist at that position. Do nothing if a plant is already there. The given position can be 0, 1, 2, or 3.

`get_name(self) -> str` *(method)*

Return the name of this room instance.

`remove_plant(self, position: int) -> Optional[Plant]` *(method)*

Return a Plant at a given position from a Pot, None if no plant exists. Removes the plant from a pot at the given position.

`progress_plant(self, pot: Pot) -> bool` *(method)*

Return True if pot is not empty and triggers a given pot to check on plant condition and plant to age. False if pot is empty.

`progress_plants(self) -> None` *(method)*

Progress each pot with the `progress_plant` method in ascending order of position. i.e. pot at position 0, pot at position 1, pot at position 2, then pot at position 3.

`__str__(self) -> str` *(method)*

Return the string representation of this room.

`__repr__(self) -> str` *(method)*

Return a string that could be copied and pasted to construct a new Room instance with the same name as this Room instance.

```
>>> room = Room('Bedroom')
>>> room.get_plants()
{0: None, 1: None, 2: None, 3: None}
>>> room.get_number_of_plants()
0
>>> pot = Pot()
>>> pot.put_plant(Plant('Rebutia'))
>>> pots = {0: pot, 1: Pot(), 2: Pot(), 3: Pot()}
>>> room.add_pots(pots)
>>> room.get_plants()
{0: Plant('Rebutia'), 1: None, 2: None, 3: None}
>>> room.get_number_of_plants()
1
>>> room.get_pots()
{0: Pot(), 1: Pot(), 2: Pot(), 3: Pot()}
>>> room.get_pot(0).look_at_plant()
Plant('Rebutia')
>>> room.get_pot(1).look_at_plant()
>>> room.add_plant(2, Plant('Cereus'))
>>> room.get_plants()
{0: Plant('Rebutia'), 1: None, 2: Plant('Cereus'), 3: None}
>>> room.get_pot(2).look_at_plant()
Plant('Cereus')
>>> room.get_number_of_plants()
2
>>> room.get_name()
'Bedroom'
>>> plant = room.remove_plant(0)
>>> plant.get_name()
'Rebutia'
>>> plant = room.remove_plant(1)
>>> print(plant)
None
>>> room.add_plant(2, Plant('Cereus'))
>>> room.add_plant(0, Plant('Rebutia'))
>>> room.get_plants()
{0: Plant('Rebutia'), 1: None, 2: Plant('Cereus'), 3: None}
>>> room.get_pot(0).set_sun_range((5, 8))
>>> room.get_pot(0).set_evaporation(1.1)
>>> room.get_pot(2).set_sun_range((6, 8))
>>> room.get_pot(2).set_evaporation(1.2)
>>> room.progress_plant(room.get_pot(0))
True
>>> room.progress_plant(room.get_pot(1))
False
>>> room.progress_plants()
>>> room.get_pot(0).look_at_plant().get_age()
2
>>> room.get_pot(2).look_at_plant().get_age()
1
>>> room.progress_plants()
>>> room.get_pot(0).look_at_plant().get_age()
3
>>> room.get_pot(2).look_at_plant().get_age()
2
>>> room.progress_plant(room.get_pot(0))
True
>>> room.get_pot(0).look_at_plant().get_age()
4
>>> room.get_pot(2).look_at_plant().get_age()
2
>>> print(str(room))
Bedroom
>>> room
Room('Bedroom')
```

OutDoor *(class)*

An OutDoor is a Room but outdoors.

`progress_plant(self, pot: Pot) -> bool` *(method)*

Returns True if pot is not empty and triggers a given pot to check on plant condition and plant to age. False if pot is empty. Checks to see if an animal attack has occured. **Note:** Make use of the *a2_support.dice_roll*() function and that *a2_support.dice_roll*() is only called once within this `progress_plant()` method and if the pot is empty, *a2_support.dice_roll*() should not be called.

```
>>> outdoor = OutDoor('Balcony')
>>> outdoor.add_plant(0, Plant('Rebutia'))
>>> outdoor.add_plant(1, Plant('Disocactus'))
>>> outdoor.get_pot(0).set_sun_range((5, 8))
>>> outdoor.get_pot(0).set_evaporation(1.1)
>>> outdoor.get_pot(1).set_sun_range((6, 8))
>>> outdoor.get_pot(1).set_evaporation(1.2)
>>> disocactus = outdoor.get_pot(1).look_at_plant()
>>> rebutia = outdoor.get_pot(0).look_at_plant()
>>> rebutia.has_repellent()
False
>>> disocactus.has_repellent()
False
>>> rebutia.set_repellent(True)
>>> rebutia.has_repellent()
True
>>> outdoor.progress_plants()
There has been an animal attack! But luckily the Rebutia has repellent.
>>> rebutia.get_health()
10
>>> disocactus.get_health()
10
>>> rebutia.get_age()
1
>>> outdoor.progress_plants()
There has been an animal attack! But luckily the Rebutia has repellent.
There has been an animal attack! Poor Disocactus.
>>> rebutia.get_health()
10
>>> disocactus.get_health()
5
```

```
>>> rebutia.get_age()
2
>>> outdoor.progress_plants()
>>> outdoor.progress_plants()
There has been an animal attack! Disocactus is dead.
>>> rebutia.get_health()
10
>>> disocactus.get_health()
0
>>> rebutia.get_age()
4
>>> disocactus.get_age()
4
>>> outdoor.progress_plants()
There has been an animal attack! But luckily the Rebutia has repellent.
Disocactus is dead
>>> rebutia.get_health()
10
>>> disocactus.get_health()
0
>>> rebutia.get_age()
5
>>> disocactus.get_age()
4
```

Model *(class)*

This is the model class that the controller uses to understand and mutate the house state. The model keeps track of multiple Room instances and an inventory. The Model class must provide the interface through which the controller can request information about the house state, and request changes to the house state.

`__init__(self, house_file: str) -> None` *(method)*

Sets up the model from the given `house_file`, which is a path to a file containing house information (e.g. `houses/house1.txt`). Once you have written the `Room` class, you can uncomment the provided `load_house` function and use it to aid in your implemention of the this method.

`get_rooms(self) -> dict[str, Room]` *(method)*

Returns all rooms with room name as keys with a corresponding room instance.

`get_all_rooms(self) -> list[Room]` *(method)*

Returns a list of all the room instances.

`get_inventory(self) -> Inventory` *(method)*

Returns the inventory.

`get_days_past(self) -> int` *(method)*

Returns the number of days since the start.

`next(self, applied_items: list[tuple[str, int, Item]]) -> None` *(method)*

Move to the next day, if there are items in the list of applied items (room name, position, item to be applied) then apply all affects. Add fertiliser and possum repellent to the inventory every 3 days. Progress all plants in all rooms. Ensure that `progress_plants` is called on each room in the same sequence rooms are given by the `load_house` function.

`move_plant(self, from_room_name: str, from_position: int, to_room_name: str, to_position: int) -> None` *(method)*

Move a plant from a room at a given position to a room with the given position.

`plant_plant(self, plant_name: str, room_name: str, position: int) -> None` *(method)*

Plant a plant in a room at a given position.

`swap_plant(self, from_room_name: str, from_position: int, to_room_name: str, to_position: int) -> None` *(method)*

Swap the two plants from a room at a given position to a room with the given position.

`get_number_of_plants_alive(self) -> int` *(method)*

Return the number of plants that are alive in all rooms.

`has_won(self) -> bool` *(method)*

Return True if number of plants alive > 50% of number from start of the 15 day period. And 15 days has passed.

`has_lost(self) -> bool` *(method)*

Return True if number of plants alive <= 50% of number from start of the 15 day period.

`__str__(self) -> str` *(method)*

Returns the text required to construct a new instance of `Model` with the same game file used to construct `self`.

`__repr__(self) -> str` *(method)*

Does the same thing as `__str__`.

```
>>> model = Model('houses/house1.txt')
>>> model.get_rooms()
{'Bal1': OutDoor('Balcony'), 'Bed1': Room('Bedroom')}
>>> model.get_all_rooms()
[OutDoor('Balcony'), Room('Bedroom')]
>>> model.get_inventory()
Inventory(initial_items=[Fertiliser(), Fertiliser(), PossumRepellent(),
PossumRepellent()], initial_plants=[Plant('Rebutia'), Plant('Cereus'),
Plant('Disocactus')])
>>> model.get_rooms().get('Bal1').get_plants()
{0: Plant('Rebutia'), 1: Plant('KingOyster'), 2: None, 3:
Plant('FiddleLeafFig')}
>>> model.plant_plant('Rebutia', 'Bal1', 2)
>>> model.get_rooms().get('Bal1').get_plants()
{0: Plant('Rebutia'), 1: Plant('KingOyster'), 2: Plant('Rebutia'), 3:
Plant('FiddleLeafFig')}
>>> model.get_inventory()
Inventory(initial_items=[Fertiliser(), Fertiliser(), PossumRepellent(),
PossumRepellent()], initial_plants=[Plant('Cereus'),
Plant('Disocactus')])
>>> model.get_rooms().get('Bal1').get_plants()
{0: Plant('Rebutia'), 1: Plant('KingOyster'), 2: Plant('Rebutia'), 3:
Plant('FiddleLeafFig')}
>>> model.get_rooms().get('Bed1').get_plants()
{0: None, 1: Plant('Cereus'), 2: Plant('Monstera'), 3:
```

```
Plant('BridgesiiMonstrose')}
>>> model.move_plant('Bal1', 0, 'Bed1', 0)
>>> model.get_rooms().get('Bal1').get_plants()
{0: None, 1: Plant('KingOyster'), 2: Plant('Rebutia'), 3:
Plant('FiddleLeafFig')}
>>> model.get_rooms().get('Bed1').get_plants()
{0: Plant('Rebutia'), 1: Plant('Cereus'), 2: Plant('Monstera'), 3:
Plant('BridgesiiMonstrose')}
>>> plant = model.get_rooms().get('Bed1').get_pot(1).look_at_plant()
>>> plant.get_name()
'Cereus'
>>> plant.get_age()
0
>>> model.has_won()
False
>>> model.get_days_past()
1
>>> model.next([])
Poor KingOyster dislikes the sun levels.
There has been an animal attack! Poor KingOyster.
Poor Cereus dislikes the sun levels.
>>> plant.get_age()
1
>>> model.has_lost()
False
>>> plant.has_repellent()
False
>>> model.next([('Bed1', 1, PossumRepellent())])
Poor KingOyster dislikes the sun levels.
There has been an animal attack! KingOyster is dead.
Poor Cereus dislikes the sun levels.
>>> plant.has_repellent()
True
```

5.2 Controller class

GardenSim *(class)*

GardenSim is the controller class, which should maintain instances of the model and view, collect user input and facilitate communication between the model and view. The methods you *must* implement are outlined below, but you are strongly encouraged to implement your own helper methods where possible.

`__init__(self, house_file: str, view: View) -> None` *(method)*

Creates a new GardenSim house with the given `view` and a new Model instantiated using the given `house_file`.

`play(self) -> None` *(method)*

Executes the entire simulation until a win or loss occurs. When the `a2.py` file is run, a GardenSim instance is created and this method is run. As such, this method should cause all of the program output. For examples of how this method should operate, see the `example_simulations/` folder, which contains example outputs of full GardenSim games.

6 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging.

6.1 Marking Breakdown

Your total grade for this assessment piece will be a combination of your functionality and style marks. For this assignment, functionality and style have equal weighting, meaning you should be devoting at least as much time towards proper styling of your code as you do trying to make it functional.

6.2 Functionality Marking

Your program's functionality will be marked out of a total of 50 marks. As in Assignment 1, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment. You may receive partial marks within each class for partially working methods, or for implementing only a few classes.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.

Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked, but if there are errors in your code that cause the interpreter to fail to execute your program, you may get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.10 interpreter. If it runs in another environment (e.g. Python 3.9 or PyCharm) but not in the Python 3.10 interpreter, you may get zero for the functionality mark.

6.3 Style Marking

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will also be out of 50.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

6.3.1 Readability

Program Structure Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.

Descriptive Identifier Names Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.) The main reason for this restriction is that most people who follow the *Hungarian Notation* convention, use it poorly (including Microsoft).

Named Constants All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.

6.3.2 Documentation

Comment Clarity Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.

Informative Docstrings Every class, method and function should have a docstring that summarises its purpose. This includes describing parameters and return values so that others can understand how to use the method or function correctly.

Description of Logic All significant blocks of code should have a comment to explain how the logic works. For a small method or function, the logic should usually be clear from the code and docstring. For long or complex methods or functions, each logical block should have an in-line comment describing its logic.

Structure will be assessed as to how well your code design conforms to good object-oriented programming practices.

6.3.3 Object-Oriented Program Structure

Classes & Instances Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.

Encapsulation Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.

Inheritance & Polymorphism Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.

6.3.4 Algorithmic Logic

Single Instance of Logic Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method or function.

Variable Scope Variables should be declared locally in the method or function in which they are needed. Attributes should be declared clearly within the `__init__` method. Class variables are avoided, except where they simplify program logic. Global variables should not be used.

Control Structures Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

6.4 Documentation Requirements

There are a significant number of classes and contained methods you have to implement for this assignment. For each one, *you must provide documentation* in the form of a docstring. The only exception is for overridden methods on subclasses, as python docstrings are inherited.

6.5 Assignment Submission

This assignment follows the assignment submission policy as assignment 1. Please refer to the assignment 1 task sheet.

You must submit your assignment as a single Python file called `a2.py` (use this name – all lower case), and *nothing else*. Your submission will be automatically run to determine the functionality mark. If you submit a file with a *different name*, the tests will *fail* and you will get *zero* for functionality. Do *not* submit the `a2_support.py` file, or any other files. Do *not* submit any sort of archive file (e.g. zip, rar, 7z, etc.).

6.6 Plagiarism

This assignment follows the same plagiarism policy is as per Assignment 1. Please refer to the assignment 1 task sheet.

7 Updates

- `decrease_water(self, amount: float) -> None` *(method)*

  Decrease the plants water by a specified amount.
- `get_drink_rate(self) -> float` *(method)*

  Return water drinking rate of the plant
- `get_water(self) -> float` *(method)*

  Return the water levels of the plant.
- house_examples should be called example_simulations
- `Inventory.get_inventory()` not required.
- `Plant` example output updated.
- `Water` example output updated. To reflect `Plant.get_water()`
- `a2_suppot.dice_roll()` updated to reflect a 15% chance.
- `Pot` example output updated.
- move set Figure 2. apply item updated.
- `get_sun_levels(self) -> tuple[int, int]` *(method)*

  Return the acceptable sun level of the plant with the lower and upper range.
- `UserInterface` changed to `View`
- update to example simulations.
- `a2.py` `Controller` updated to `GardenSim`.
- `Model.has_loast()` should say <= 50%.

- `Plant.drink_water()` should check if the water level equal or less than 0.
- `a2_support.dice_roll()` should only be called once in `OutDoor.progress_plant()`. Ensure that `progress_plants` is called on each room in the same sequence rooms are given by the `load_house` function. i.e. if the order was `['Bal1', 'Bed1']` then `Bal1.progress_plants()` should be called first, then `Bed1.progress_plants()`. Also, make sure that `Room.progress_plants()` progresses the pot in ascending position order. i.e. progress pot in position 0, 1, 2, 3 respectively.