



---

COMP1046

Object-Oriented Programming

## **Assignment Part 2**

## **OO Programming**

Prepared by Michael Ulpen

March 2024

---

## ASSIGNMENT PART 2 LEARNING OUTCOMES

---

Part 2 of this assignment will develop your skills by allowing you to put into practice what has been taught in class during the first 8 weeks of the course.

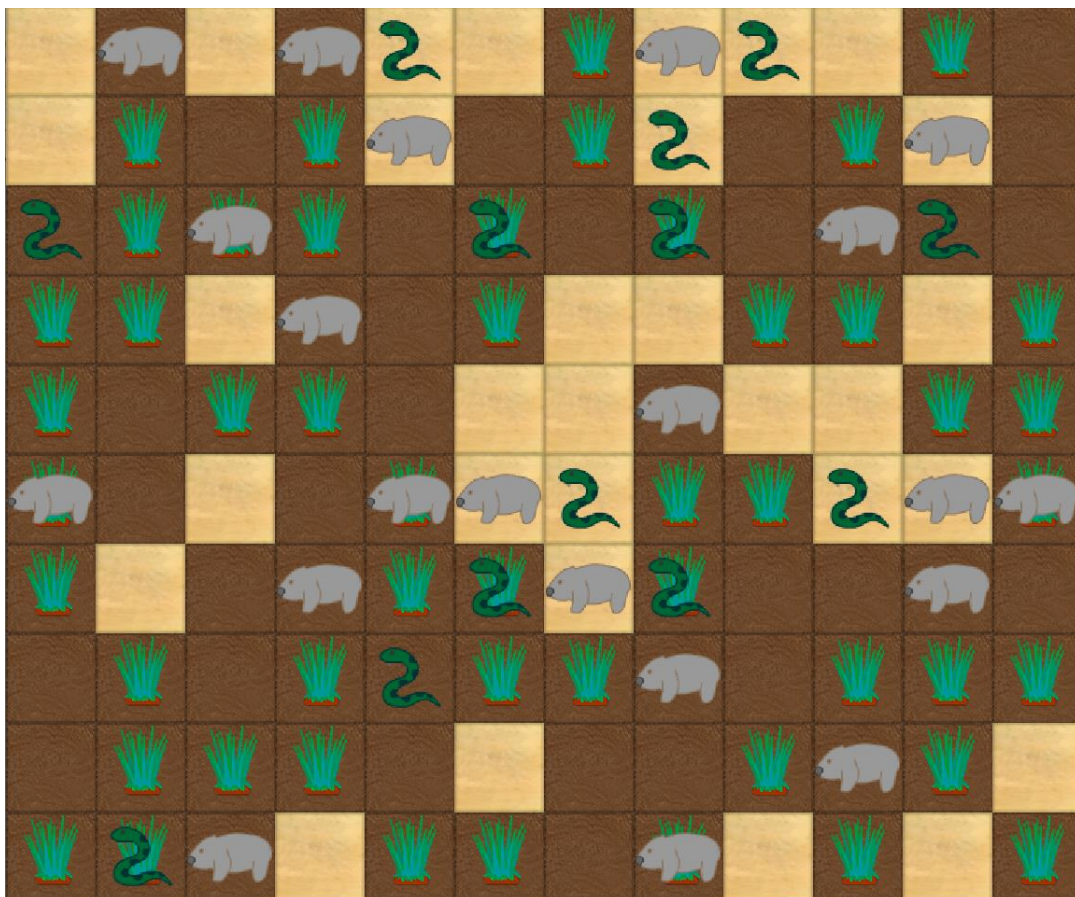
- Implement an object-oriented software design in Python.
- Apply object-oriented principles including abstraction, data hiding, encapsulation, inheritance, and polymorphism to software implementation.
- Practice Python programming skills including commenting with docstrings, handling exceptions, and version control with git.

---

## INTRODUCTION

---

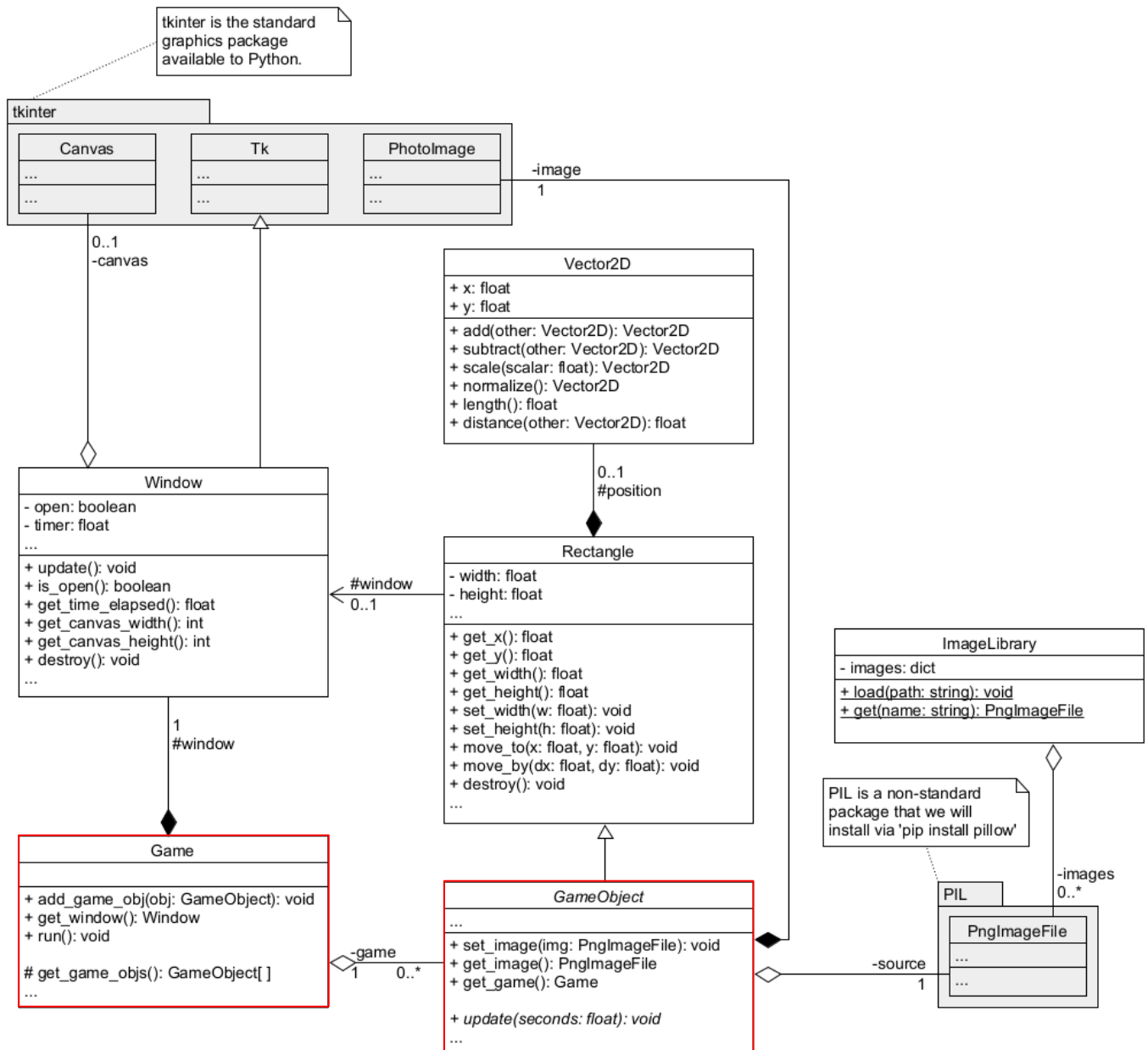
This assignment will require you to design, implement, test, and debug a graphical ecosystem simulator called EcoSim. You will need to use classes in a provided game engine, write classes representing animals, land tiles, and plants, and apply appropriate relationships between them. In addition, the assignment will require you to document your classes appropriately and use version control to keep track of your progress.



The assignment is split into two parts: OO Design (due week 6) and **OO Programming (due end of week 12)**. Part 1 of this assignment (OO Design) is to be performed in groups of 3-4 students and will be presented by students in the workshop class. **Part 2 of this assignment (OO Programming) is to be performed individually and submitted online. This document gives instructions for Part 2 (OO Programming).** If you are unsure about any instructions in this document, please seek clarification from your lecturer.

## PART 2 – OO PROGRAMMING

The assignment files include a game engine, called **game.py**, which will automatically handle graphics and object updates. Part of the assignment is learning how to use the provided game module to create a graphical application. There are several classes in **game.py** and many methods in those classes. The following UML diagram shows the design of **game.py**, including the most important attributes and methods in each class and how those classes relate to each other.



Note that **game.py** has been updated since Assignment Part 1. You will need to modify your design to incorporate the above classes. The methods and classes in *italics* are abstract methods, and underlined methods are class/static methods.

Note that you will need to use pip to install some packages before you can use **game.py** or start the assignment. Please read the PIL section for details.

Game.py is a module that will be available with part 2 of the assignment. Please read the following for an explanation of the classes and methods that may be accessed from game.py:

### **Game**

EcoSim should extend from Game. If you extend from Game, your class will automatically contain a Window and an aggregation of GameObjects. You may add a GameObject to the Game by calling `add_game_obj(obj)`. You may get a list of all added GameObjects by calling the protected method `_get_game_objs()`. Please do not call this method unless you absolutely need to. You should create and use your own aggregation relationships instead e.g., a list of animals and tiles in EcoSim. If you call the inherited `run()` method, it will start a loop which will automatically update the graphics in the Window and will call `update()` on all GameObjects.

### **GameObject**

*GameObject replaces the Image class from Assignment Part 1.*

Animal, Plant, and Tile should inherit from GameObject. GameObject is a subclass of Rectangle, and therefore has a width, height, and position. It also contains an Image, which it will draw at its position. It is associated with a Game and will draw itself to that Game's window. The GameObject's image may be resized and moved using methods inherited from Rectangle. Each GameObject has a method called `get_game()` that returns the Game it is attached to.

### **Window**

When a Window is constructed, it will open a window with the given width and height (default 1152x964). The graphics drawn to the window are intended to be moved around and updated. We will generally update the graphics within a loop that looks like the following:

```
window = game.Window()
timer = 0

while window.is_open():
    timer = window.get_time_elapsed()

    # move and update game objects

    window.update()
```

The while loop will iterate as quickly as possible while the window is open. Closing the window will cause `is_open()` to return False, ending the loop.

`Window.get_time_elapsed()` tells us how many seconds have passed since the last iteration of the loop. This number is usually a very accurate floating point number between 0 and 1 seconds. The timer can be useful if we want graphics to move or update slowly. For example, if you want to trigger an event after 60 seconds, you may check if the timer is greater than 60, trigger the event, and reset the timer to 0.

We must call `window.update()` as the last line in the loop. This method is responsible for repainting the screen and taking input events. If we do not call this method, the program will freeze and be unresponsive.

### **Rectangle**

Rectangles contain a width, height, and a position of type `Vector2D`. If a Window is provided to the constructor, they will be drawn to the screen. They can be moved using `move_to(x, y)`, which moves the Rectangle to that position. They may also be moved using `move_by(dx, dy)`, which moves the Rectangle by a given amount in a direction represented by dx and dy.






## ImageLibrary

ImageLibrary contains only static methods. These methods must be called using the class name, ImageLibrary. The load method takes a folder path as an argument and automatically loads all .PNG image files in that folder. It adds these images as values to a dictionary using the file name (without extension) as the key. You may get PngImageFiles from the ImageLibrary using their file name.





A folder called 'images' is provided with the assignment files. It is important to call ImageLibrary.load('images') as the first line in your program. This ensures the Images in that folder are available to use. For example:

```
ImageLibrary.load('images')
image = ImageLibrary.get('wombat1')
```

The following files are available for use:

wombat1	snake1	grass_tuft	dirt_tile	sand_tile
				

There are several other files that are **not necessary** to use. You could use them if you wanted to animate the animals or implement additional land types.

wombat2	snake2	grass_tile	water_tile
			

*It will be necessary to add your own images to this folder for your additional animal and plant class.*

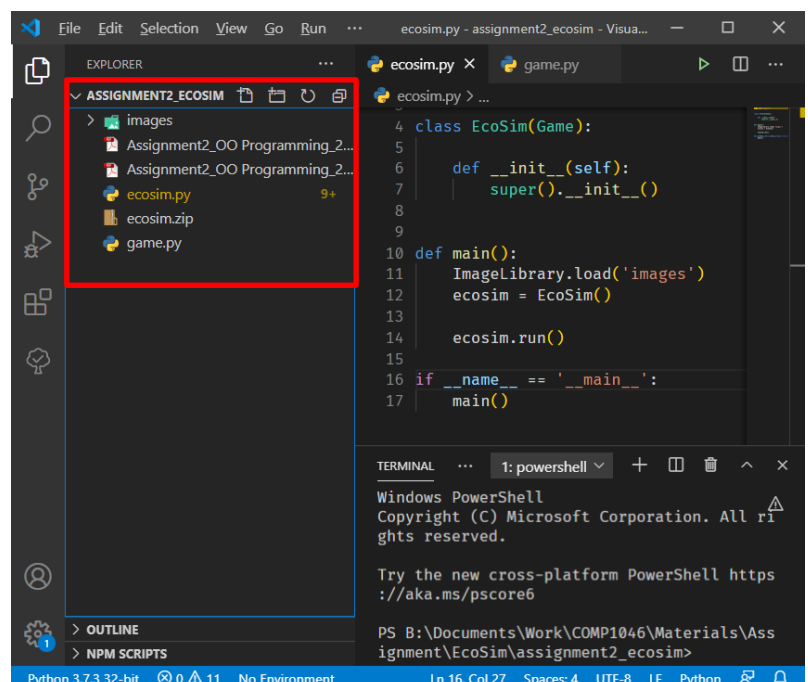
**WARNING:** You must only have your assignment folder open in Visual Studio Code. If you do not open your assignment folder in Visual Studio Code using File -> Open Folder... -> ecosim, you may receive errors such as "**KeyError: 'sand\_tile'**". This is caused by Visual Studio Code looking in the top-level folder for the "images" folder.

## PIL

ImageLibrary relies on code provided by the PIL library, also known as pillow. PIL is used to load PNG images. You will need to install the pillow package using **one** of the following commands:

```
pip install pillow
pip3 install pillow
python -m pip install pillow
python -m pip install --user pillow
python3 -m pip install --user pillow
```

If none of these commands work, please talk to your lecturer.

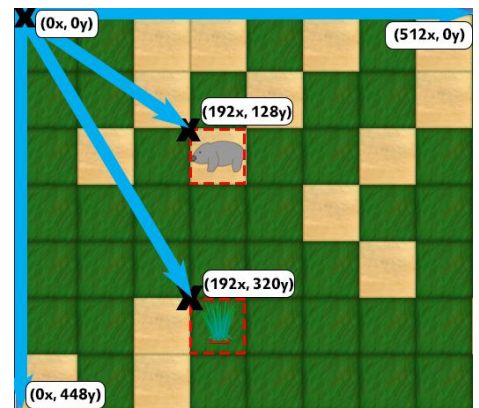




## Vector2D

A vector in physics has a length and direction. The direction and length may be represented by an arrow pointing from 0,0 to x,y.

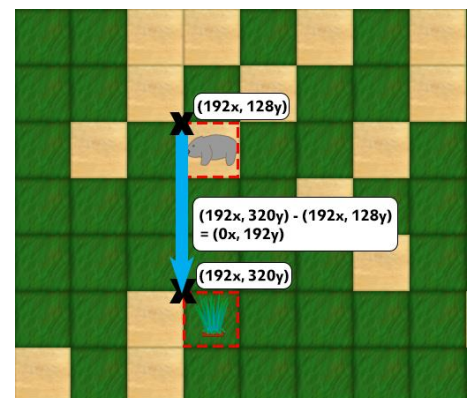
The Vector2D class may be constructed with an x and y coordinate and can be used to represent position. For example, the Wombat is at position 192, 128. This can be represented as an arrow pointing from 0, 0, and can be stored as a Vector2D position.



This class can also be used to represent a direction and speed. For example, the Wombat wants to eat the Grass. To calculate the direction to the Grass, the Wombat subtracts their position from the Grass' position:

$$(192, 320) - (192, 128) = (0, 192)$$

The Wombat must move 192 along the y axis to reach the Grass.

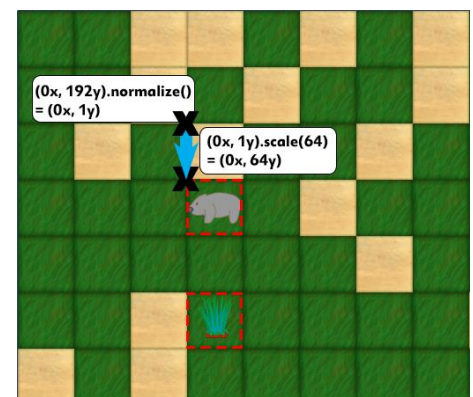


This class can also be used to represent a speed and direction at the same time. For example, the Wombat wants to move towards the Grass but may only move very slowly.

The normalize method forces the Vector to have a length of 1. The scale method will multiply both coordinates by a given amount. We use a combination of these methods to set the length of the Vector.

$$(0, 192).normalize() == (0, 1)$$
$$(0, 1).scale(64) == (0, 64)$$

This Vector may be used to move the Wombat by that amount in that direction: `wombat.move_by(vector.x, vector)`



## Tkinter

The game.py module is mostly built from the tkinter package, which is always available to Python. This package includes the classes Tk, Canvas, and PhotoImage. You will not directly use tkinter in this assignment, but you will need to use game.py.

---

## EcoSIM

---

EcoSim attempts to simulate the relationship between predators and prey in the environment. Each animal will have to find food to survive. If they survive for long enough, they will reproduce to create more animals. Some animals eat other animals. Some animals eat plants. In this assignment, snakes will eat the wombats and the wombats will eat grass. The wombats will breed faster than the snakes but will have to eat more to stay alive.

In this assignment, you will need to define at least 12 classes: EcoSim, Tile, DirtTile, SandTile, Animal, Wombat, Snake, Plant, Grass, OutOfBoundsException, and another Animal and Plant of your choice. You have already been provided with the files game.py, and ecosim.py.

---

## GETTING STARTED

---

In Visual Studio Code, go to File/Open Folder... and select the folder containing ecosim.py, game.py and the images folder. Ensure these are the **only files** visible from Visual Studio Code. Initialize a git repository in this location and make your first commit.

Open ecosim.py and you will notice the following code:

```
from game import *
```

```
class EcoSim(Game):  
    def __init__(self):  
        super().__init__()
```

```
def main():  
    ImageLibrary.load('images')
```

```
    ecosim = EcoSim()  
    ecosim.run()
```

```
if __name__ == '__main__':  
    main()
```

First, we import from game.py.

This allows access to the classes Game, ImageLibrary, and other classes in that file.

EcoSim inherits methods from Game. When an EcoSim is constructed, it will call super().\_\_init\_\_(), which automatically creates and opens a Window.

The main function is the starting point for the program. ImageLibrary loads the .PNG files from the images folder, which allows you to access them using ImageLibrary.get(name).

EcoSim inherits the run() method from Game. Calling run() will start a loop that automatically updates the graphics in the EcoSim's window.

The if statement checks that ecosim.py is being run as a program - not being imported as a module. If ecosim is run as a program, it will call the main function, which starts the program.

Run this file and you will notice an empty window. Close the window to end the program.

In the EcoSim class heading, hold CTRL on your keyboard and click Game. This should automatically open game.py and navigate to the Game class. Read the code in class Game. Notice that it contains a list called \_gameObjects. Read the code in the run() method. Notice that it updates \_window in a while loop.

*The following is an example of how to create a class and add it to the Game. You should write similar code to implement each of your animals, plants, and tiles.*

Add the following code to `ecosim.py`, above the `EcoSim` class:

```
class Tile(GameObject):  
    def __init__(self, position, width, height, sourceImage, game):  
        super().__init__(position, width, height, sourceImage, game)
```

The `Tile` class extends from `GameObject`, which inherits its methods. `GameObject.__init__()` requires 5 arguments: `position` is a `Vector2D`; `width` and `height` are floats; `sourceImage` is an `Image` returned from the `ImageLibrary`; `game` is any object that inherits from `Game`.

In the `main()` function, **above** the line `ecosim.run()`, construct a `Tile`:

```
tile = Tile(Vector2D(0, 0), 96, 96, ImageLibrary.get('water_tile'), ecosim)
```

Run the code and you should see the following error message:

```
TypeError: Can't instantiate abstract class Tile with abstract methods update
```

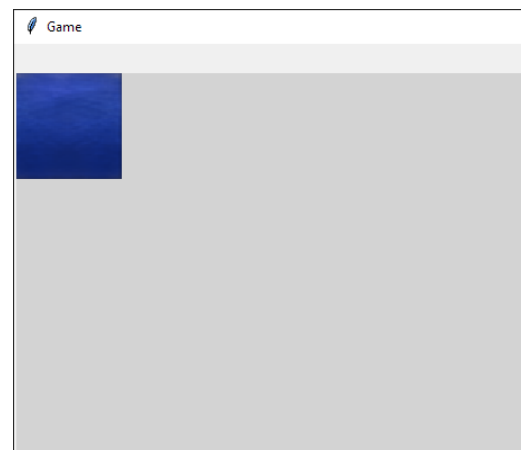
To fix this error, implement the abstract update method, inherited from `GameObject`, in the `Tile` class:

```
def update(self, timeElapsed):  
    print('Time Elapsed =', timeElapsed)
```

Note that if you get a different error message, it may mean you did not install PIL or you did not open your assignment folder in Visual Studio Code correctly. Also note that code written **below** `ecosim.run()` will not be executed until the window is **closed**.

Run the code and you should see a 96x96 water tile at position 0, 0. If you look at the terminal, you may see output such as:

```
Time Elapsed = 0.0009915828704833984  
Time Elapsed = 0.0  
Time Elapsed = 0.0010020732879638672
```



This demonstrates that the `Tile`'s update method is being called automatically. The last output shows that 0.001 seconds have passed since the last time the object was updated.

Play around with this code by changing arguments to the `Tile` constructor, or by changing the print statement in the `update()` method. You may hold CTRL and click on



Construct another tile object **within** `EcoSim.__init__()`:

```
self.tile = Tile(Vector2D(0, 96), 96, 96, ImageLibrary.get('grass_tile'), self)
```

Notice the following changes:

- The y coordinate is 96.
- The image is 'grass\_tile'.
- The game argument is self (the EcoSim being constructed).

When you feel you understand this code, ***please delete these two Tile objects***; they will not be needed. You should modify and inherit the Tile class to create DirtTile and SandTile.

***Your main() function must only contain:***

```
def main():  
    ImageLibrary.load('images')  
    ecosim = EcoSim()  
  
    ecosim.run()
```

All your code must be written within classes. You must correctly implement association, aggregation, and composition relationships to allow your classes to communicate and share information. EcoSim is the starting point of your program. Construct your other objects within the methods of that class.

***The following is an example of how to create an Animal class and update it.***

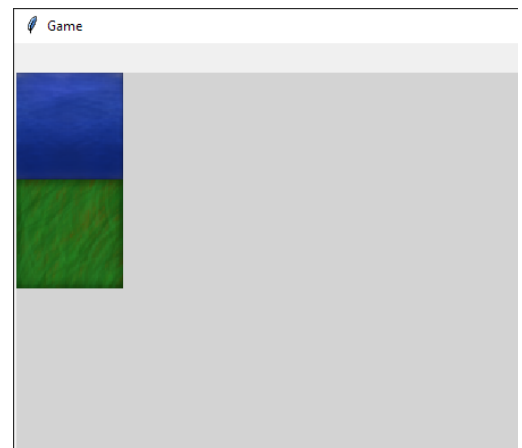
```
class Animal(GameObject):  
    def __init__(self, position, width, height, sourceImage, game, speed, energy):  
        super().__init__(position, width, height, sourceImage, game)  
  
        self.speed = speed  
        self.energy = energy  
  
    def update(self, timeElapsed):  
        self.energy -= timeElapsed  
  
        if self.energy <= 0:  
            self.destroy()  
  
        else:  
            self.move_by(self.speed, self.speed)
```

Other than the arguments required by its super constructor, Animal's constructor sets two other attributes: speed and energy.

The update method subtracts the time from self.energy (a fraction of a second).

If self.energy reaches zero, the Animal will be removed from the window, and from the Game.

This will cause the Animal to move by (speed, speed) every update step, if it is alive.



Construct an Animal in EcoSim such as:

```
self.animal = Animal(Vector2D(96, 96), 96, 96, ImageLibrary.get('wombat1'), self, 10, 5)
```

Run the program. You should notice the Animal moves very *fast*. Slow it down by scaling the speed by the time:

```
self.move_by(self.speed * timeElapsed, self.speed * timeElapsed)
```

The above code means the Animal moves by (10, 10) pixels every second and dies after 5 seconds. To make the Animal move towards a target location, we need to use the Vector2D class. Create a target in the `__init__` method:

```
self.target = Vector2D(random.randint(self.get_x()-50, self.get_x()+50),  
                        random.randint(self.get_y()-50, self.get_y()+50))
```

This target is a random position within a 100 pixel wide square around the Animal's position. You may move towards this location by modifying the update method. **Delete** the line that calls `self.move_by` and **replace** it with:

```
trajectory = self.target.subtract(self.get_position())  
trajectory = trajectory.normalize().scale(self.speed * timeElapsed)  
self.move_by(trajectory.x, trajectory.y)
```

Run the program again. You will notice the Animal moves towards a random location, stops, and then dies.

Write a for loop in the EcoSim `__init__()` method to fill a list with 20 Animals such as:

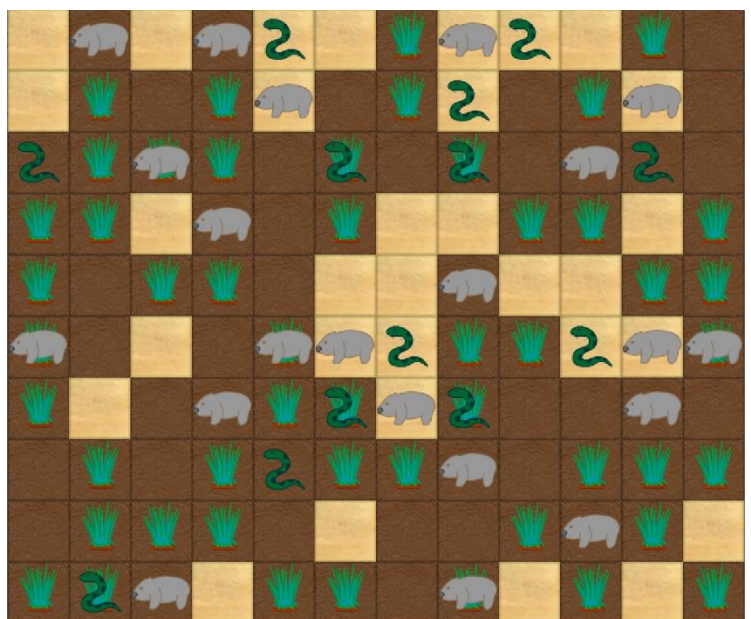
```
self.animals = []  
for i in range(20):  
    animal = Animal(Vector2D(96, 96), 96, 96, ImageLibrary.get('snake1'), self, 10, 5)  
    self.animals.append(animal)
```

Run the program again. You will notice many Animals moving towards a random locations before they stop and then die.

When you feel you understand this code, ***please delete these Animals***. You should not use the Animal class. You should modify and inherit it to create Wombat and Snake.

You should now have the understanding and the tools necessary to start this project. You may like to write your code in these steps:

- Write the Tile classes
- Write a nested loop to create a grid of random Tiles.
- Grow Grass on some Dirt Tiles.
- Add Animals in random locations.
- Modify the Animal's update() methods to move, findFood, breed, and die.
- Modify the Grass to spread.
- Add an additional Plant and Animal class and integrate them into the game.



---

## REQUIREMENTS

---

You may need to use/modify your design from Assignment Part 1. Please implement the following classes:

### EcoSim

You must have a class that represents the simulation as a whole. The EcoSim will create the window and fill it with a 12x10 grid of land tiles. Each land tile will need to be separated by 96 pixels. Each land tile will be a random one of DirtTile or SandTile. Randomly, around 50% of the time, a DirtTile will start with a Grass on top. It should spawn an appropriate number of Wombats and Snakes. You may try 10 Wombats and 3 Snakes and adjust the numbers to balance the ecosystem.

EcoSim must be the class that runs the program, for example:

```
def main():
    ImageLibrary.load('images')
    ecosim = EcoSim()

    ecosim.run()

if __name__ == '__main__':
    main()
```

### DirtTile

Dirt tiles should have a width and height of 96 pixels. Dirt tiles will be randomly generated in a grid along with sand tiles. At the beginning of the simulation, dirt tiles have a 50% chance to spawn with a grass on top. Dirt may only have one grass growing on it at a time. The dirt tile will need to know both its screen position (multiple of 96), and its grid position (0-12, 0-10).



### SandTile

Sand tiles should have a width and height of 96 pixels. Sand tiles will be randomly generated in a grid along with dirt tiles. Sand tiles may not grow grass. The sand tile will need to know both its screen position (multiple of 96), and its grid position (0-12, 0-10).



### Grass

Ensure Grass descends from GameObject so it inherits its methods and variables. Grass is eaten by wombats and only grows on top of dirt tiles. After a random number of seconds between 5 and 20, a grass may spread to create a new grass on top of an adjacent dirt tile, but only if that dirt tile does not already have a grass growing on it. You may use the type() or isinstance() functions to check if a GameObject is a DirtTile. When a grass is eaten, it should be removed from the dirt tile and then destroyed.



## Wombat

Wombats eat grass and are eaten by snakes. When they are low on energy, they will target the closest nearby grass and move towards it. When they are within 48 pixels of the grass location, the grass is destroyed, and the wombat recovers some energy. You may recover 10 energy and adjust for balance.



When the Wombat is not hungry, it will move towards a random nearby location, choosing a new location whenever it reaches within 48 pixels of its target. The new location must be created in a method called `selectTarget()`. `selectTarget()` should generate a random `Vector2D` within a 100 pixel square around the Wombat. If the new target is outside the Window, it should raise an `OutOfBoundsException`, otherwise it should set this as the Wombat's target. When this Exception is caught in the `update()` method, it should print the Exception's error message and call the method again. The bounds of the window is 12\*96 pixels wide, and 10\*96 pixels tall. If the Wombat is still alive after 18 seconds, it will reproduce to create a new Wombat at the same location. If the Wombat runs out of energy, or is eaten by a Snake, it will die.

## Snake

Snakes eat Wombats. When they are low on energy, they will target the closest Wombat and move towards it. When they are within half a tile width of their target Wombat, they will kill the Wombat and recover their energy.



When the Snake is hungry, it moves faster, at 45 pixels per second. When the Snake is not hungry, it moves at 20 pixels per second to a random nearby location, choosing a new location whenever it reaches within 48 pixels of its target. The new location must be created in a method called `selectTarget()`. `selectTarget()` should generate a random `Vector2D` within a 100 pixel square around the Snake. If the new target is outside the Window, it should raise an `OutOfBoundsException`, otherwise it should set this as the Snake's target. When this Exception is caught in the `update()` method, it should print the Exception's error message and call the `selectTarget()` method again. The bounds of the window is 12\*96 pixels wide, and 10\*96 pixels tall. After 32 seconds, if the Snake is still alive, it will reproduce to create a new Snake at the same location.

## OutOfBoundsException

`OutOfBoundsException` should inherit from `Exception`. Its constructor should take an error message as an argument and pass it to the super constructor.

---

## DOCSTRINGS

---

Please provide a docstring on the first line of each of your classes and methods. Docstrings are written in triple quotes on the first line **inside** a class or method, such as:

```
class EcoSim(Game):
    """
    A class to represent the Ecosystem Simulator.
    """

    def __init__(self):
        """
        Constructing an object of this class will create Window containing a grid of
        Tiles and animals at random locations.
        """
        pass
```

Your docstrings may be short and simple. Document your code as you go, not all at the end. Write a brief comment about the one thing that the method is supposed to do before you write the code, then refer to that comment while implementing it. This helps maintain focus when implementing the method and should stop you from making it do more than the one thing it is supposed to do.

---

## HINTS

---

The following code samples may help you:

- You may calculate the distance between a GameObject and a Vector2D using:

```
distance = self.get_position().distance(vec2D)
```

- You may calculate the distance between two GameObject's using:

```
distance = self.get_position().distance(obj.get_position())
```

- From any GameObject, you may get all the GameObjects that have been added to the game using:

```
objects = self.get_game()._get_game_objs()
```

You will lose some marks for using this method. Only use this method if you cannot access these objects by creating and using your own aggregation/composition relationships.

- You may need to check the type of an object using:

```
isWombat = type(object) == Wombat
```

Or...

```
isAnimal = isinstance(object, Animal)
```

---

## OTHER REQUIREMENTS

---

Your code must fulfill the following requirements:

- You must demonstrate multiple examples of **inheritance**.  
*You may need to include classes that were not mentioned to demonstrate appropriate inheritance relationships.*
- You must demonstrate multiple examples of **aggregation/composition/association**. *You may need to create lists of animals or tiles in the EcoSim class.*
- You must create methods not mentioned in the text above.
- You must include **one more** type of **animal**, and one more **plant** in your program.
- You must demonstrate polymorphism by calling overridden methods, or by checking the types of objects.
- You must demonstrate good encapsulation by making dangerous variables private or protected and providing getter or setter methods to them.
- You must include `__str__` and `__repr__` methods for each class. *You may `print(self)` or `print(animals)` to help you with debugging.*
- You must update your UML diagram written in part 1 and submit it with your code.
- Your code must be documented appropriately using docstrings.
- You must make at least **10 commits** using **git** with appropriate comments.  
*You must use git version control to keep track of your progress during implementation. Only a local git repository is required in this assignment. You are not required to use an online repository, but if you choose to, please make sure your online repository is private. You should perform regular commits as you implement features and fix errors. Your commit comments should be short and reflect the changes that were made.*



---

## SUBMISSION DETAILS

---

You are required to submit an electronic copy of your program via Moodle. Make sure your submission includes:

- .git folder
- image folder
- Python files (named appropriately)
- Updated UML diagram as a .PNG image.

---

## EXTENSIONS AND LATE SUBMISSIONS

---

There will be **no** extensions/late submissions for this course without one of the following exceptions:

1. A medical certificate is provided that has the timing and duration of the illness and an opinion on how much the student's ability to perform has been compromised by the illness. **Please note** if this information is not provided the medical certificate WILL NOT BE ACCEPTED. Late assessment items will not be accepted unless a medical certificate is presented to the Course Coordinator. The certificate must be produced as soon as possible and must cover the dates during which the assessment was to be attempted. In the case where you have a valid medical certificate, the due date will be extended by the number of days stated on the certificate up to five working days.
2. A SAIBT councillor contacts the Course Coordinator on your behalf requesting an extension. Normally you would use this if you have events outside your control adversely affecting your course work.
3. Unexpected work commitments. In this case, you will need to attach a letter from your work supervisor with your application stating the impact on your ability to complete your assessment.
4. Military obligations with proof.

Applications for extensions must be lodged with the Course Coordinator before the due date of the assignment.

Note: Equipment failure, loss of data, 'Heavy work commitments' or late starting of the course are not sufficient grounds for an extension.

---

## ACADEMIC MISCONDUCT

---

Students are reminded that they should be aware of the academic misconduct guidelines available from the SAIBT website.

Deliberate academic misconduct such as plagiarism is subject to penalties. Information about Academic integrity can be found in the "Policies and Procedures" section of the SAIBT website.

---

## MARKING CRITERIA

---

Object Oriented Programming (COMP1046) Assignment Part 2 - Weighting: 15 %			
Name:	Max	Mark	Comment
<b>Correct Output:</b> Dirt, Sand, Wombat, Snake, Grass, Other, move, eat, reproduce, die.	32		
<b>Inheritance:</b> At least 3 super classes and 7 subclasses.	10		
<b>Associations:</b> At least 5 appropriate examples of other relationships e.g., association, composition, aggregation. -2 marks for each use of <i>self.get_game().get_game_objs()</i> .	10		
<b>Method Overriding:</b> +2 marks for <code>__str__()</code> and <code>__repr__()</code> methods; +2 marks for <code>update()</code> methods; And at least 3 other examples of method overriding.	10		
<b>Polymorphism:</b> Performs type checking or uses overridden methods.	10		
<b>Exception Handling:</b> <code>OutOfBoundsException</code> defined, raised, and caught correctly.	5		
<b>Documentation:</b> All classes and methods have brief doc comments.	8		
<b>Git Commits:</b> At least 10 git commits, with appropriate comments, and <code>.git</code> folder included.	10		
<b>Updated UML Diagram:</b> UML Diagram reflects code.	5		
<b>Total</b>	100		

### Possible deductions:

- Programming style:* Up to -10 marks for poor variable names, poor indentation, use of `break` or `return` to stop loops, use of `continue`, use of recursion, misuse of `self`, or calling object-methods as class-methods.
- Submitted incorrectly:* Up to -20 marks if assignment is submitted incorrectly. See the Submission Details section.
- Misusing game.py:* -10 marks for modifying or not importing and using the `game` module.
- Plagiarism/Collusion/AI Generation:* This assignment must be completed using your own ideas and code. You may lose 50-100% for submitting code similar to another student or from another source.