



Hello everyone and welcome to our course on game development with Python and Pygame! During this course, we will build a simple crossy rpg style game using the **Pygame library and written in Python**. As we build the game, we will explore and use **various components of Pygame** and hone our Python knowledge. We take care to explain exactly why we choose the methods that we do throughout the course while teaching best practices and how to structure a project. By the end, you will have a nifty little game to put in your portfolio and will feel more comfortable writing Python programs! The image below demonstrates what the final product will look like:



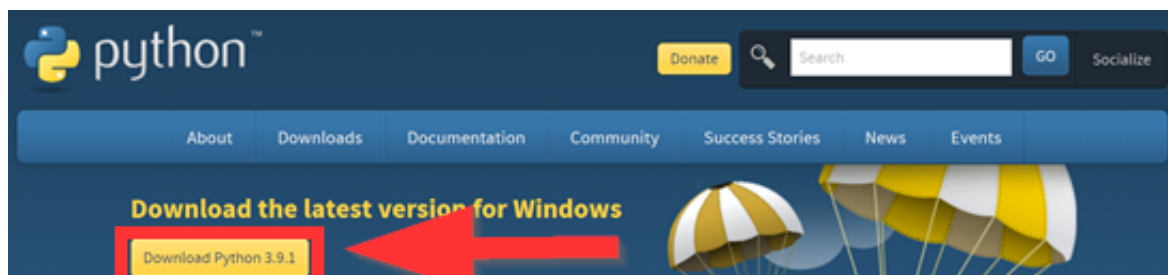
To help us build the game, we're going to need to download a version of Python, a code editing software, and the Python game library: Pygame. At this time, the latest version of Python is 3.9.0 so we will download and use that.

Pygame is a software library that contains all of the necessary components to build and run games written in Python. At this time, the latest stable release version is 1.9.6. We will need to obtain these items before writing game code.

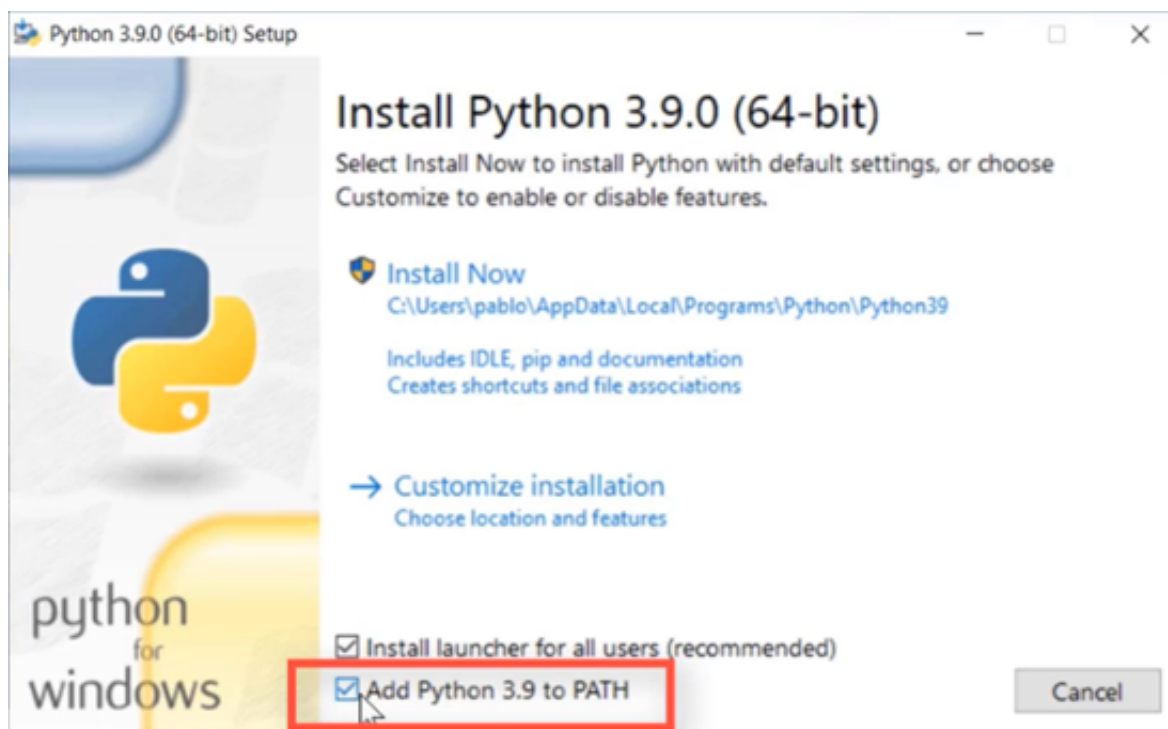
In this lesson, we're going to install all the components we'll need for our project. This includes: **Python**, **Pygame**, and **Visual Studio Code**. This lesson will focus specifically on **Windows** users, so Mac users should skip to those relevant lessons.

Python Installation

To begin, head to the **Python website** (<https://www.python.org/downloads/>) to access the Python downloads page. On this page, you can hit the yellow **Download Python** button to download the .exe for Python installation.

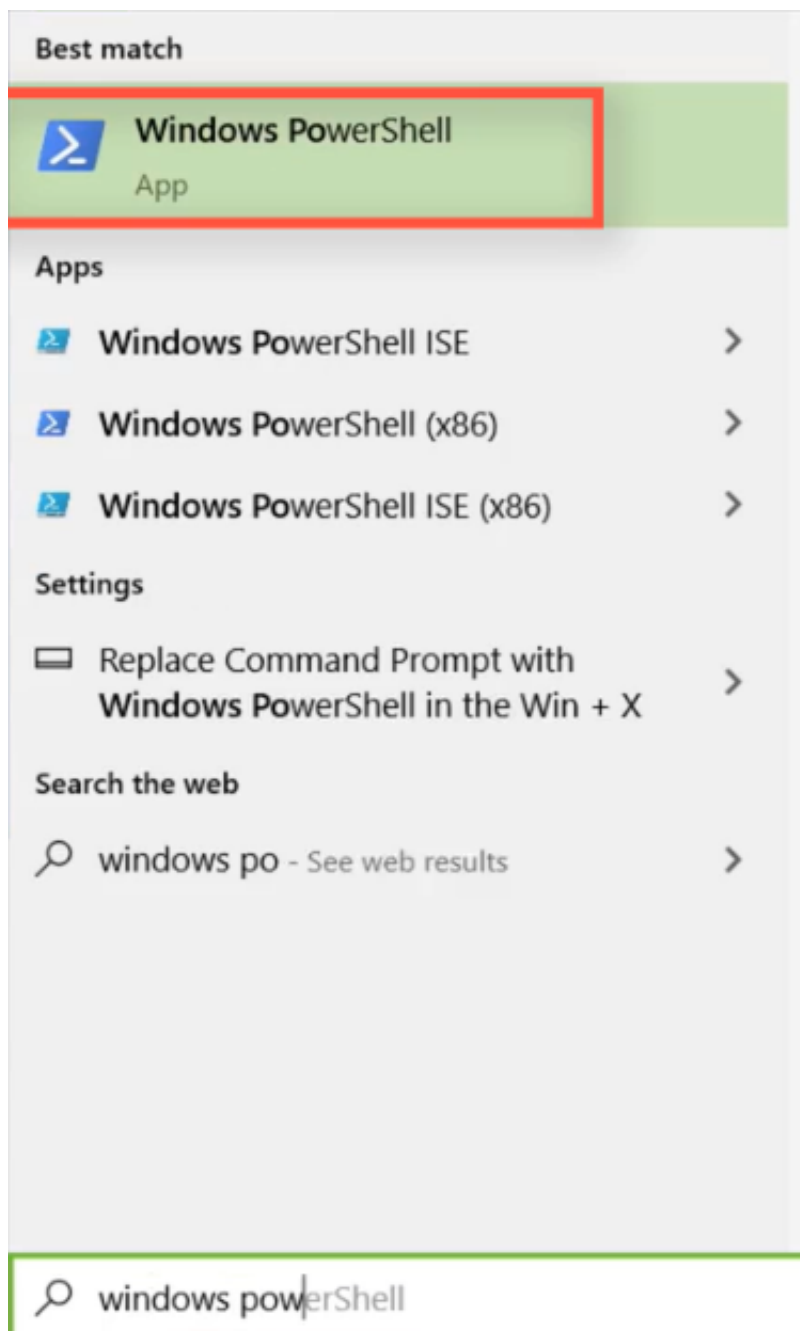


Once the .exe downloads, click the file to open the setup window. Before hitting the **Install Now** option, make sure to select the option for **Add Python to PATH**. This will ensure any IDEs on your system have access to the program.

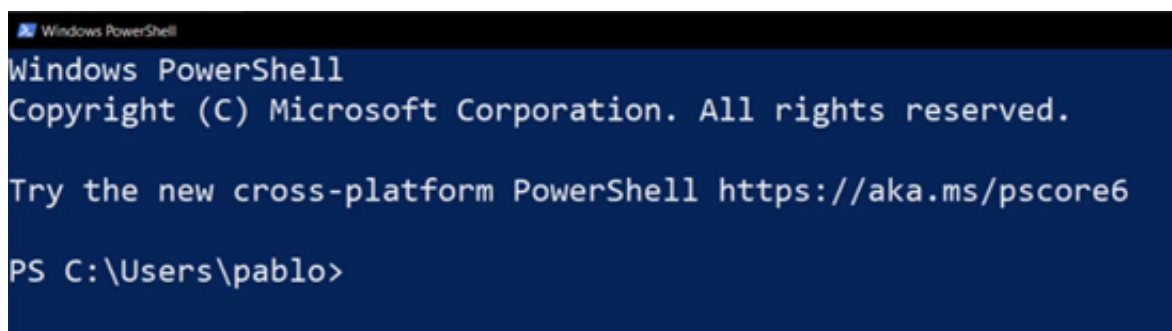


With the **PATH** option checked, hit **Install Now** to install Python. This process will take just a few minutes. However, before we move on, we'll want to test and make sure our Python installation is working.

To do so, open up the search bar and search for **Windows Powershell**.



Select the program to open it, and you'll be presented with a screen similar to Command Prompt.



To test our Python installation, we'll want to open an **Interactive Python Shell** which will allow us

python

This should display the Python version information.

```
PS C:\Users\pablo> python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

With the shell running, we can run a simple print statement and verify that it functions appropriately.

```
print("hello world! Python installed :) ")
```

```
>>> print("hello world! Python installed :) ")
hello world! Python installed :)
```

Now that we know it works, we can type the following to exit the interactive shell:

```
exit()
```

Installing Pygame

Next, we're going to install Pygame. This can be done with a simple command in Windows PowerShell:

```
pip install pygame
```

The installation may take a few moments, but once completed, you should see a notice mentioning the successful installation.

```
PS C:\Users\pablo> pip install pygame
Collecting pygame
  Downloading pygame-2.0.0-cp39-cp39-win_amd64.whl (5.1 MB)
    | ████████████████████ | 5.1 MB 2.2 MB/s
Installing collected packages: pygame
Successfully installed pygame-2.0.0
```

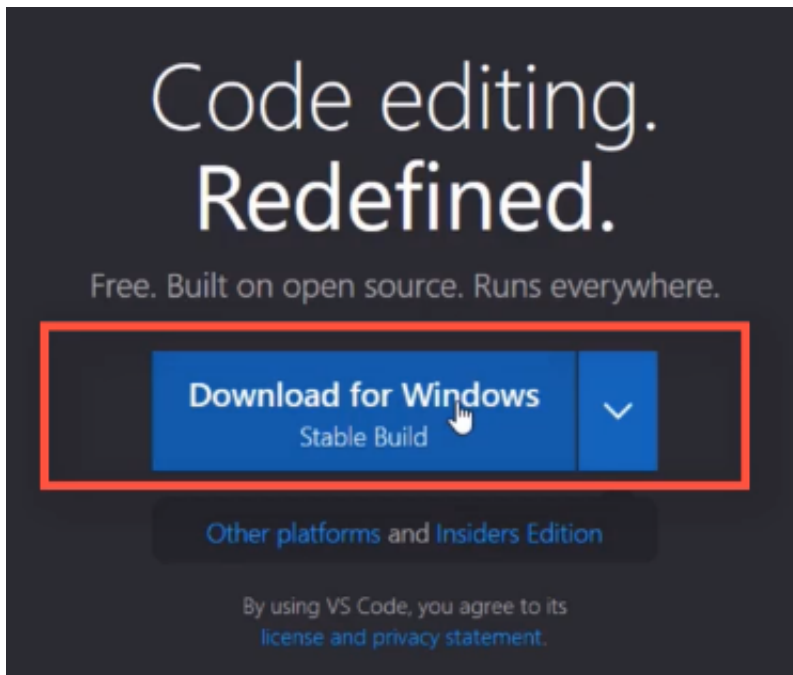
For more information about Pygame, we encourage you to visit their website:

<https://www.pygame.org/news>

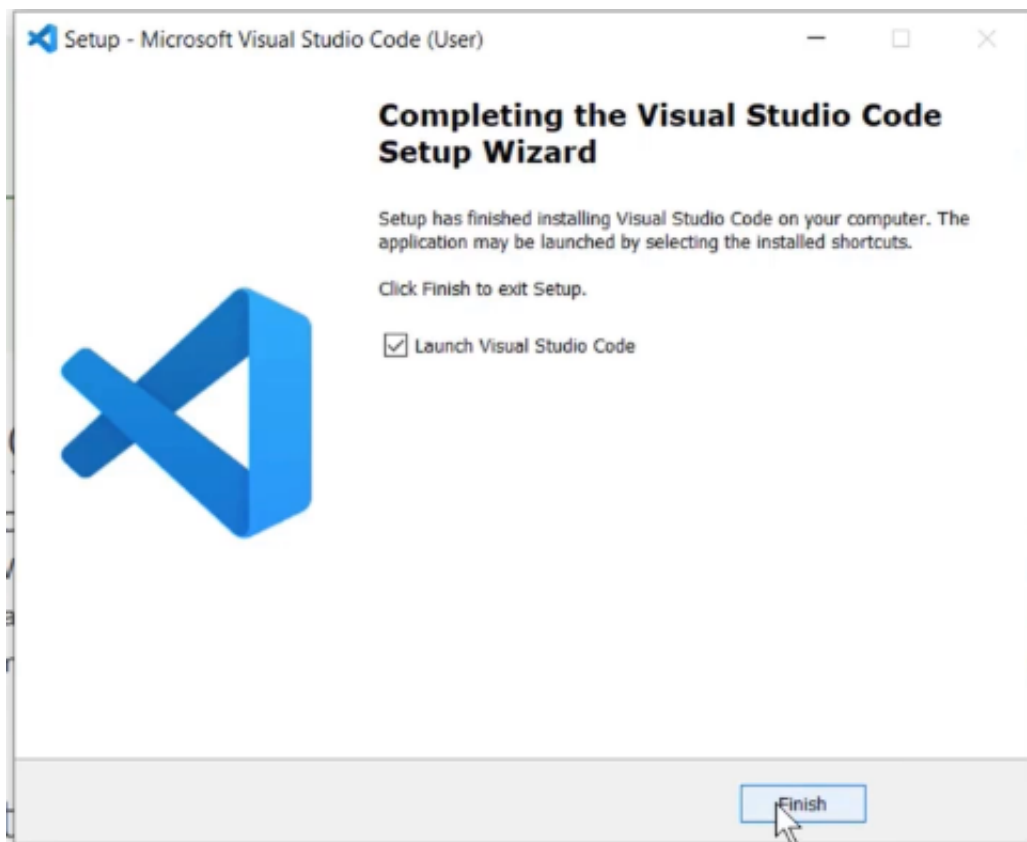
Installing Visual Studio Code

The last thing we'll do here is install Visual Studio Code (VSC) - the IDE we'll be using for this course.

Head to <https://code.visualstudio.com/> and click the big blue **Download** button to download the .exe for Visual Studio Code.



Once downloaded, you can click the .exe to open the Installation Wizard. We'll be keeping the default settings here, so simply accept the agreement and click **Next** until the installation begins.

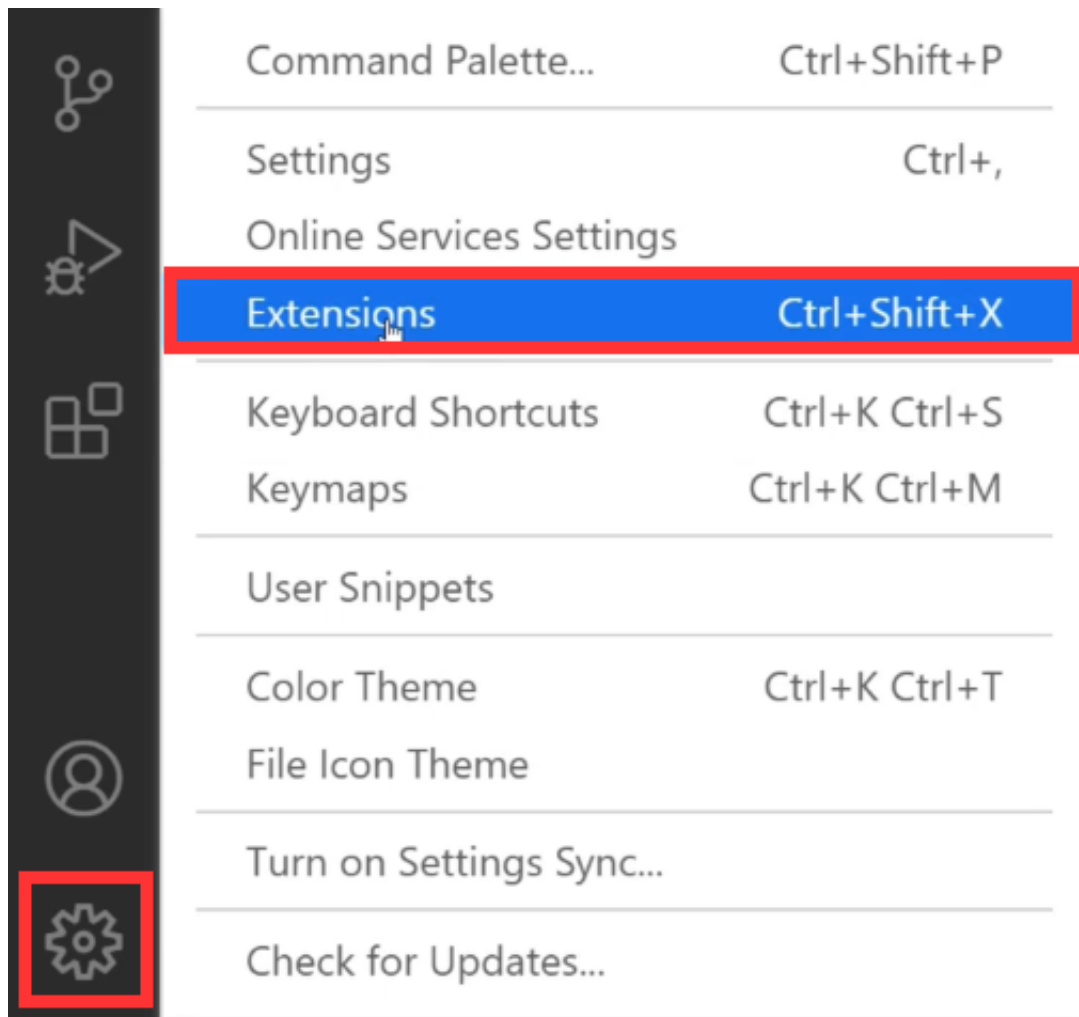


Installing Python Extension

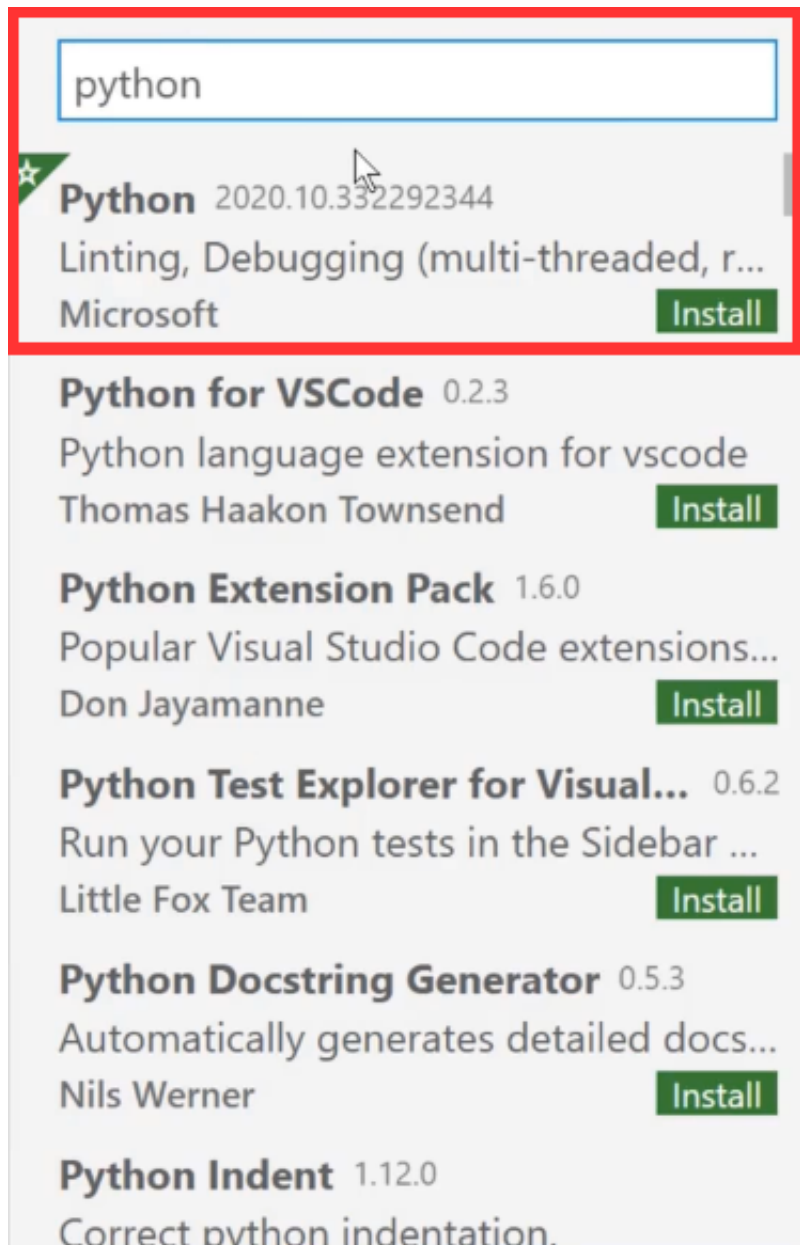


Now that Visual Studio Code is installed, we'll need a **Python Extension** in order to use it properly.

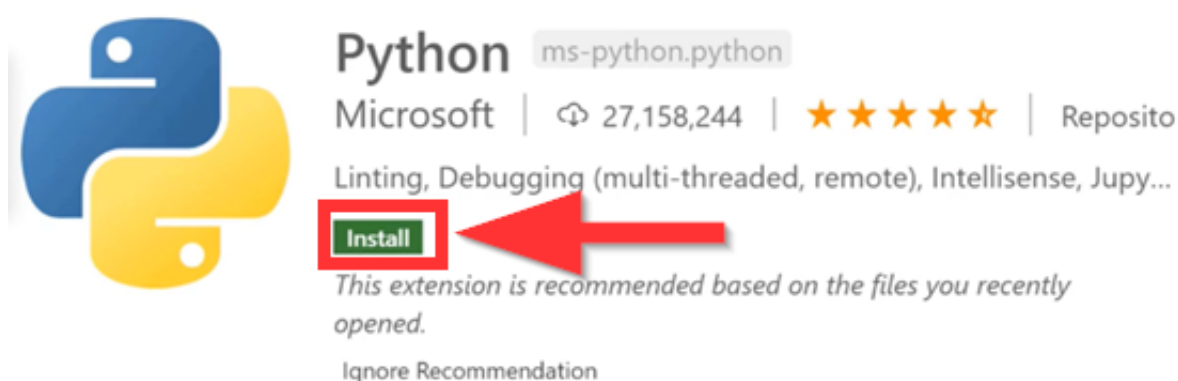
In Visual Studio Code, click the **Gear Icon** in the bottom left, and then click the **Extensions** option (alternatively, you can hit Ctrl + Shift + X).



From the Extensions window, type **python** into the search bar and select the first option.

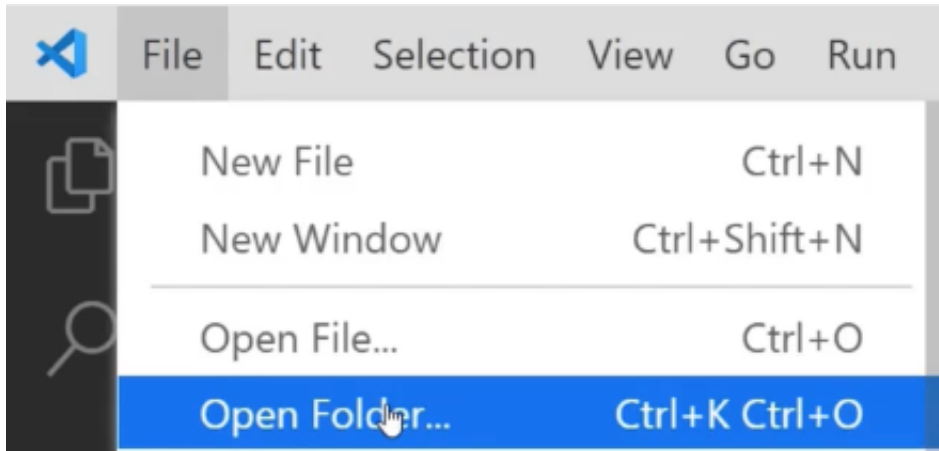


You can then hit the **Install** button to install the extension.

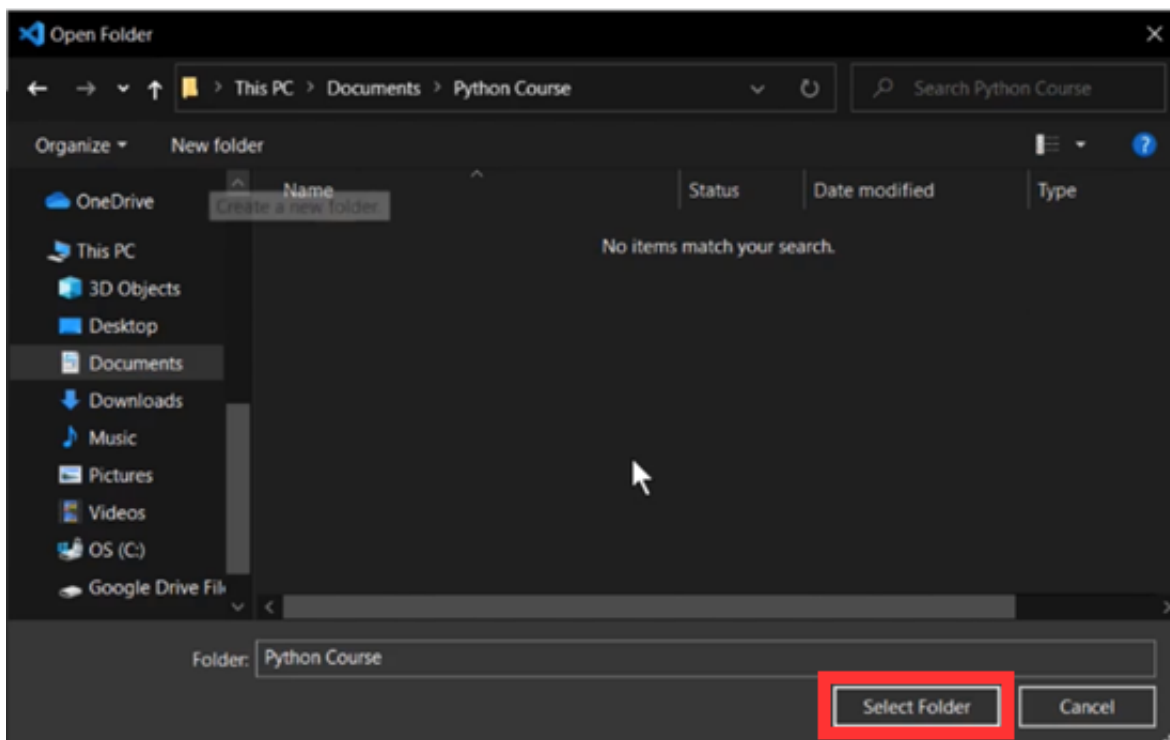


Setting Up the Project

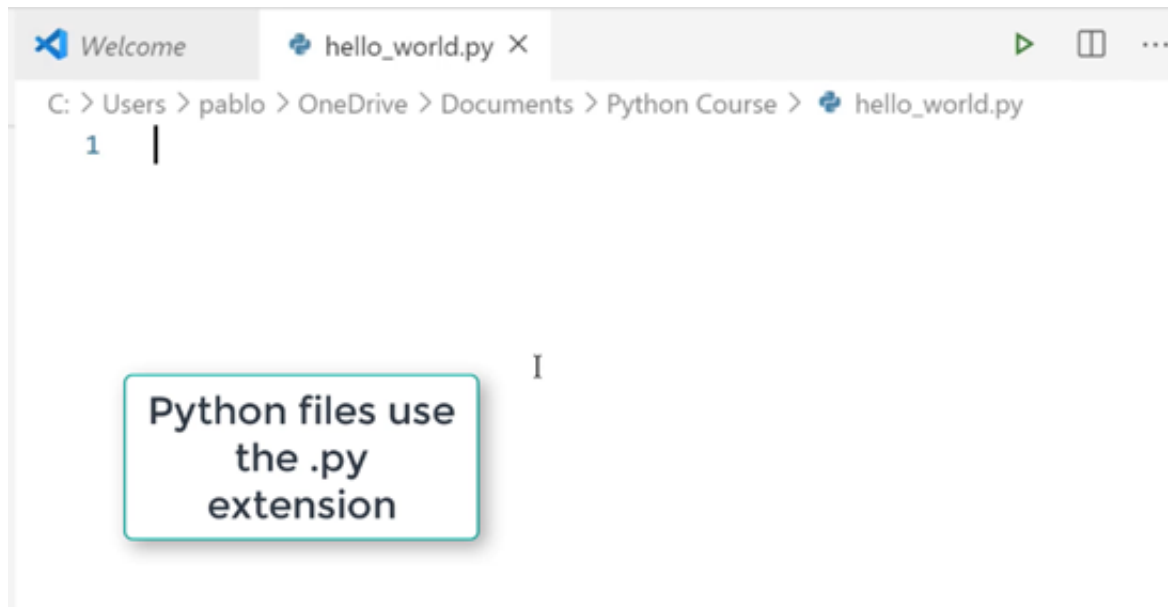
Everything is installed, so let's quickly set up a location for our project and test our installations. Head to **File** and then select **Open Folder...**



This will open the File Explorer. Create a **New Folder** and name it whatever you would like (we chose **Python Course**). Once created, you can hit **Select** folder to open it up.



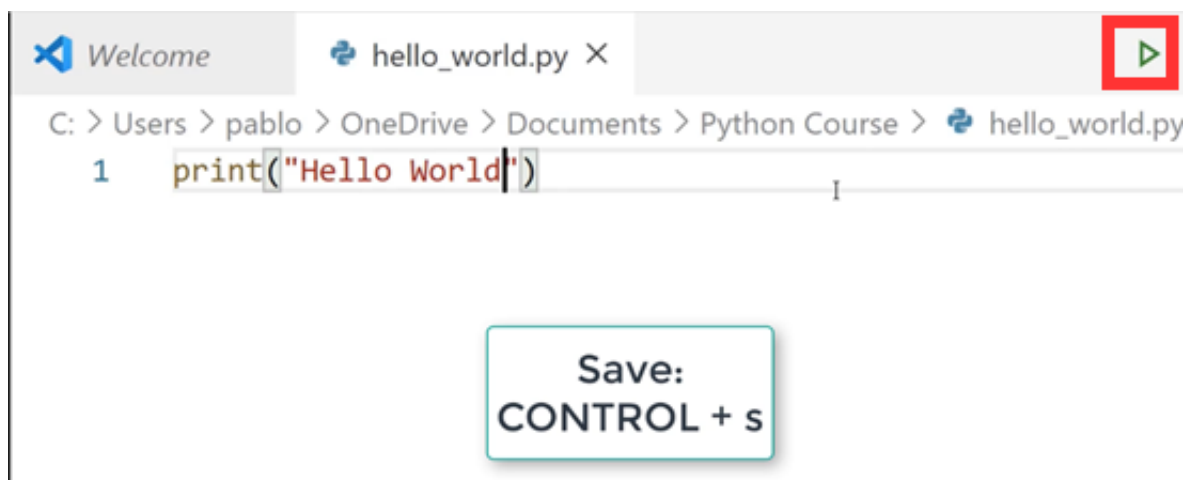
With the folder opened in Visual Studio Code, hit **File** and then **New File** to create a new file. Save the file as **hello_world.py** to convert it to a Python file.



Let's test out our file and make sure it's working. In this file, add the following:

```
print("Hello World")
```

Save the file and then click the **Play button** at the top to run the code.



A Terminal window will open in the program and show your print statement!



```
PROBLEMS TERMINAL ... 1: Python + [ ] [ ] ^ X

Copyright (C) Microsoft Corporation. All rights reserved.

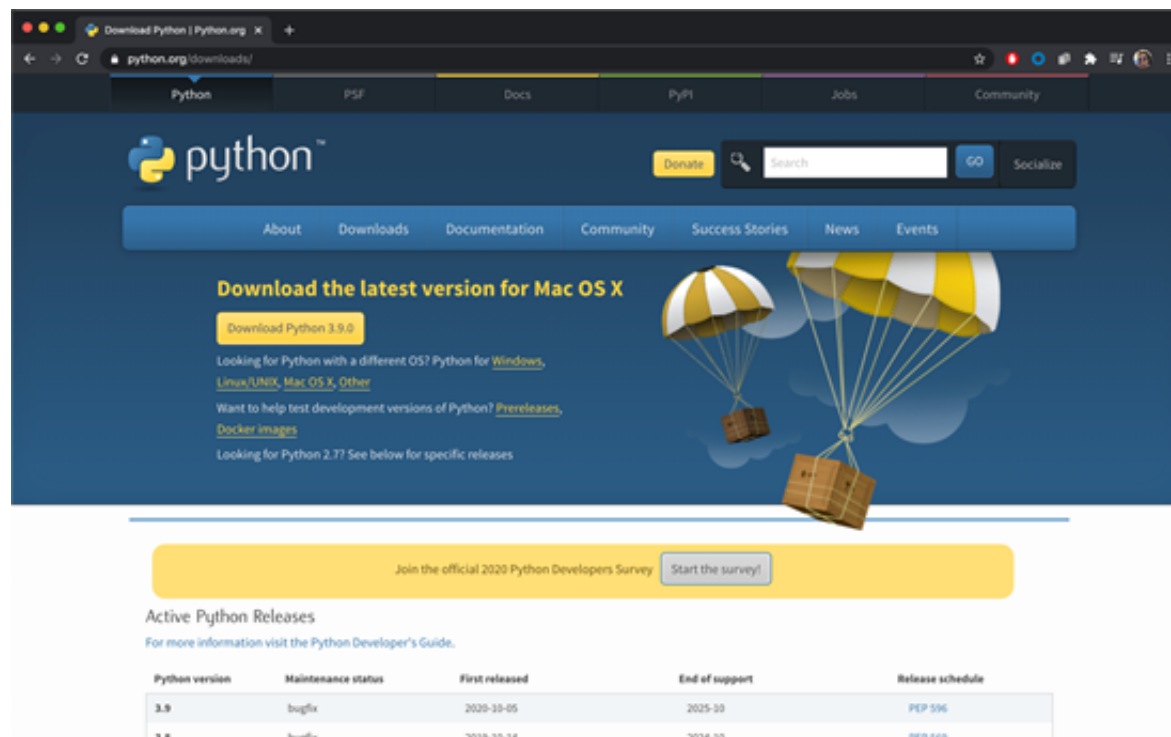
Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pablo\OneDrive\Documents\Python Course> & C:/Users/pablo/AppData/Local/Programs/Python/Python39/python.exe "c:/Users/pablo/OneDrive/Documents/Python Course/hello_world.py"
Hello World
PS C:\Users\pablo\OneDrive\Documents\Python Course>
```

We're now ready to start our project!

Most computers come with a version of Python already installed but it likely isn't the latest one. We want version 3.9.0 or whatever the latest current version is. Start by navigating to the Python downloads site:

<https://www.python.org/downloads/>



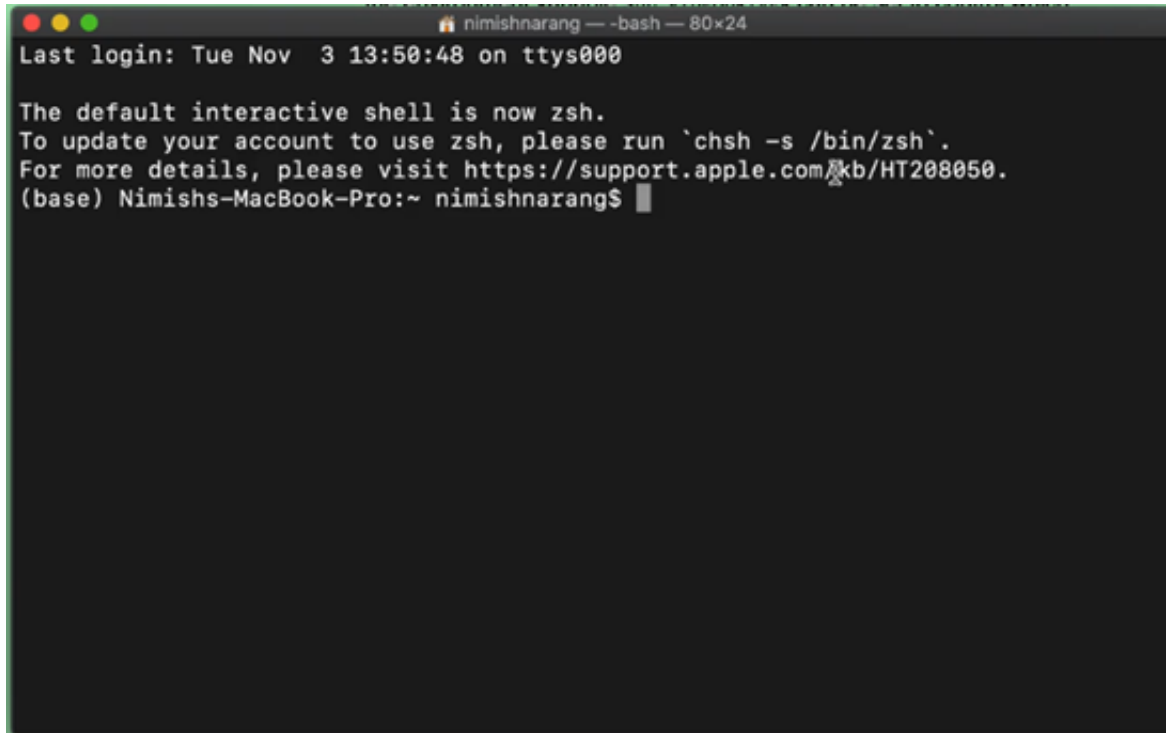
Download the appropriate version of Python for your system by clicking on the yellow button. If you want a different version, you can scroll down the page and select a different version to download.

Open the installer and follow the installation steps, selecting the default values for each step. By the end, you will be able to import this latest version into whatever development environment you choose!



The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

The Python library Pygame makes game development easy by providing means to **create screens and display images, handle user input, run game loops, and so much more**. It's also very easy to install and we can do so with a single line of code! To begin for Mac users, startup Terminal.



```
nimishnarang — bash — 80x24
Last login: Tue Nov  3 13:50:48 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) Nimishs-MacBook-Pro:~ nimishnarang$
```

To proceed, type the following command in the terminal.

The following code has been updated, and differs from the video:

```
python3 -m pip install -U pygame --user
```

This should install Pygame onto your system. Feel free to check out the Pygame website at this link:

<https://www.pygame.org/news>

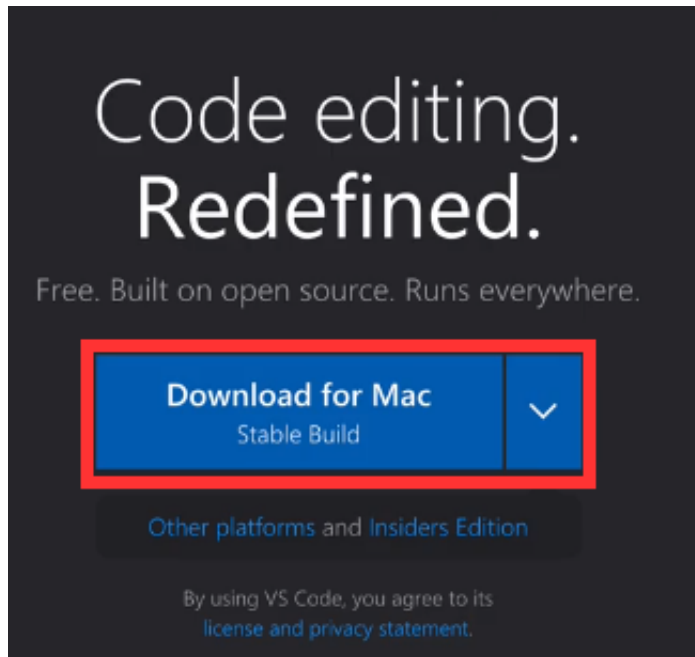
The developer documentation is particularly useful and deep dives into every component of Pygame. If you ever want to know more about a specific part of the Pygame library, simply go here and search up the item of interest:

<https://www.pygame.org/docs/>

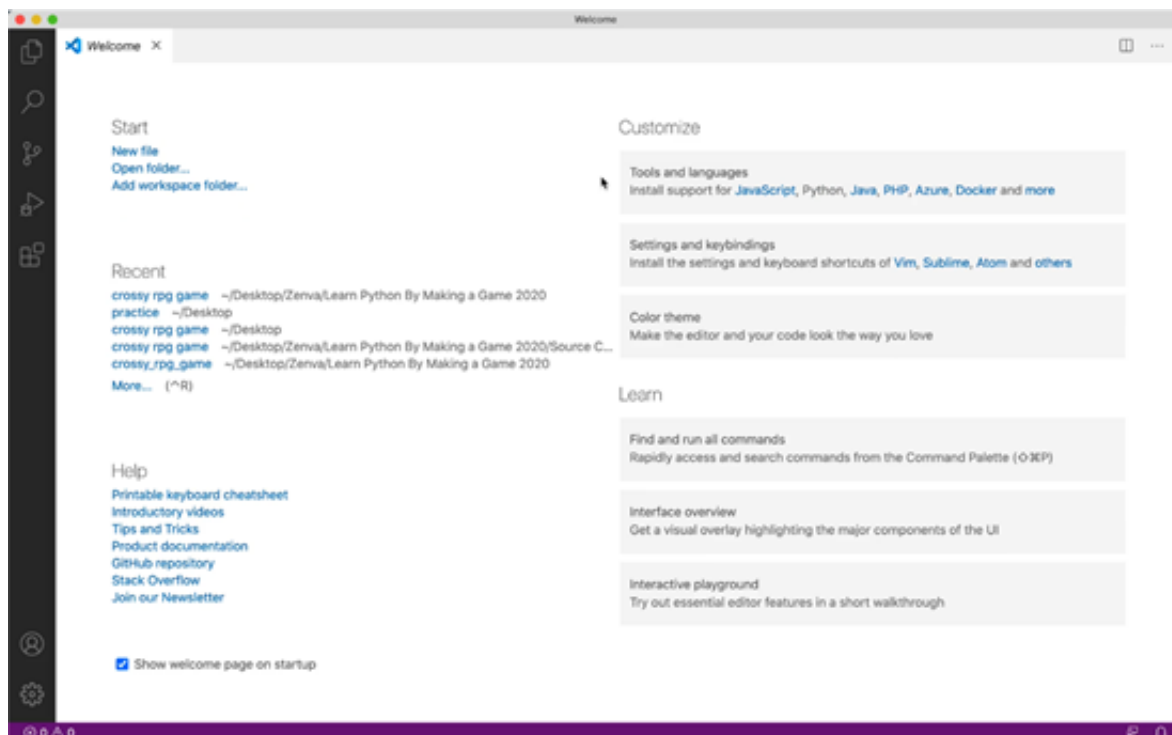
In order to edit code, we'll need a code editor or an **IDE**. For this course, we'll be using **Visual Studio Code**. This lesson will walk **Mac** users in the installation process.

Installing Visual Studio Code

To begin, head to <https://code.visualstudio.com/> and hit the big blue **Download** button to download the program (your operating system should be selected by default).

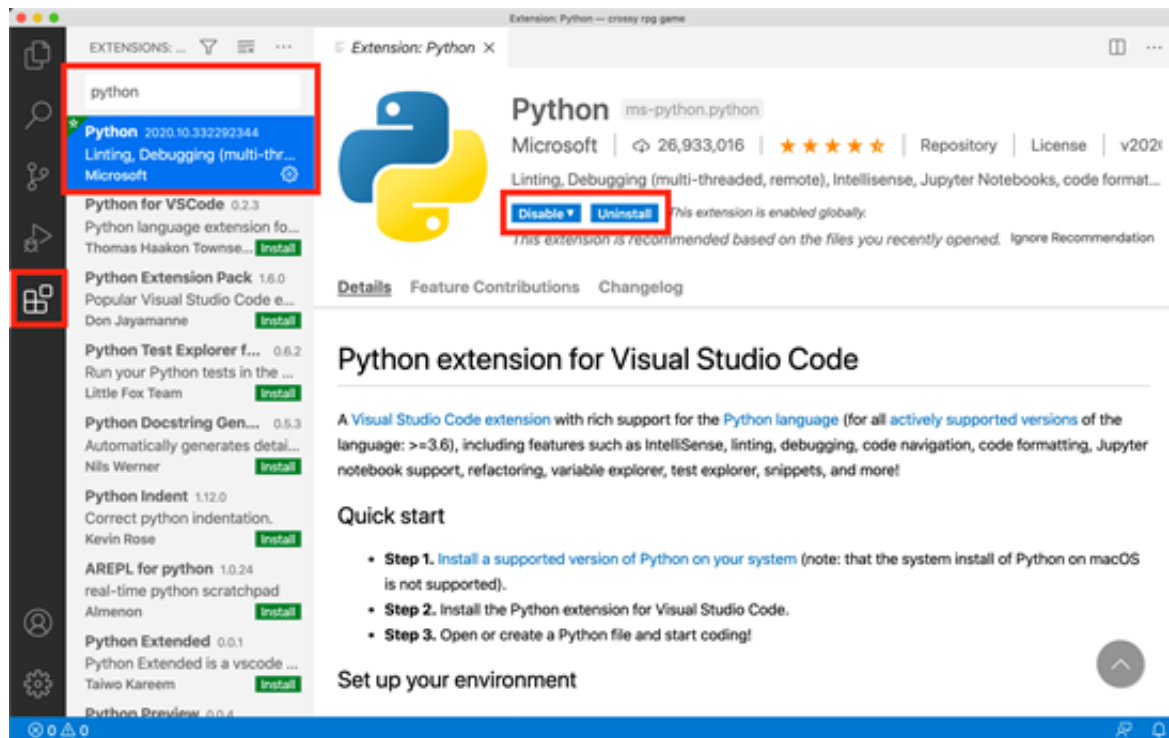


Once the download completes, click the file to **Unzip** it, which will unpackage the .app file. You can then click the .app file to open Visual Studio Code!



Python Extension

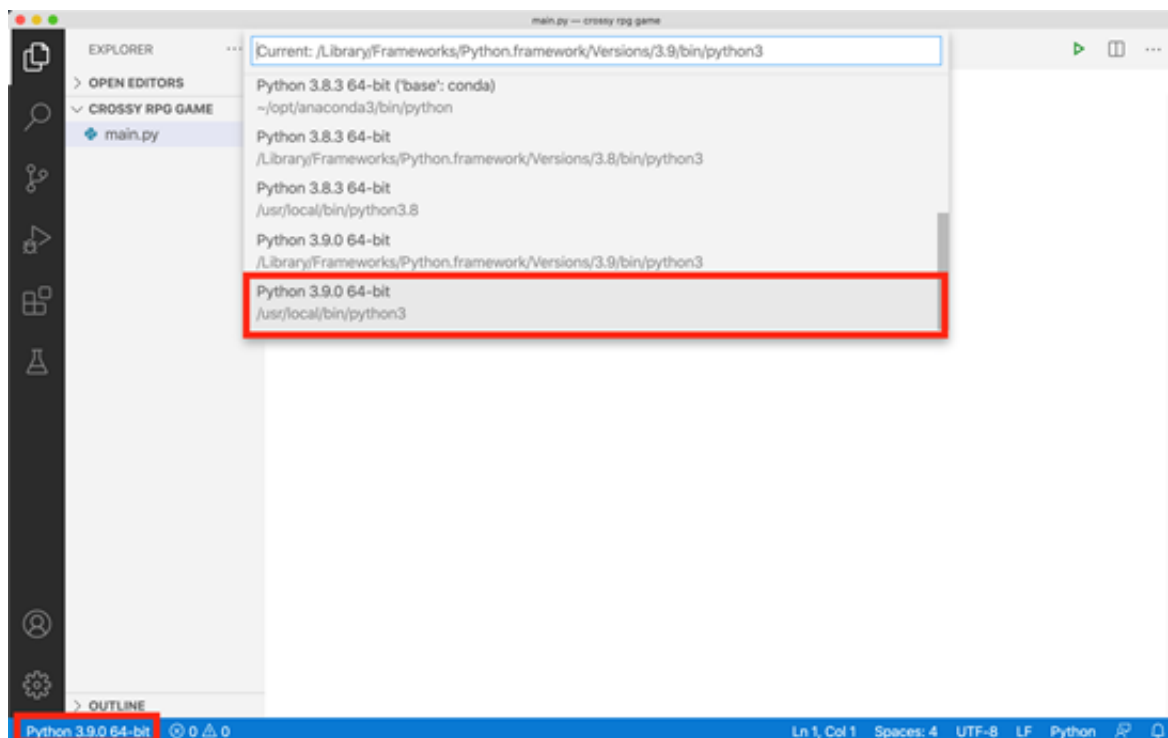
Next, we will install a **Python plugin that allows us to write and debug Python code** and adds features such as autocomplete and compile time code checking. Click on the extensions button (the 5th button from the top on the far left side shaped), type in Python in the search bar, and select the official Microsoft Python extension. Install it by clicking on install (mine says disable or uninstall because I have already installed the extension. You may need to restart VSC after the installation is complete.



Using Visual Studio Code

By this point, VSC is ready to use. To demonstrate how to use it, we will create a practice project folder, write some code to a main.py file, and run it. Start by creating a new folder in Windows Explore/Finder and call it **“practice”**. Now, turn back to VSC and select File -> Open. Find and **select the folder that we just created** and it will open the folder in VSC.

Next, create a new file by selecting File -> New File and save it as **“main.py”** by selecting File -> Save As. This **automatically converts the file type to Python** because we added the extension .py. You can **select a different version of Python by clicking on the version in the bottom left**. We recommend that you use Python 3.9.0, or whichever the latest version is at this time:



This creates a **new file** within the project folder; you will see a new file added in both VSC's File Explorer Window and in Finder/Windows Explorer. In the main.py file write this code:

```
print("Hello world!")
```

Then run the file through the VSC terminal by selecting **Terminal -> New Terminal**. This opens the terminal window at the bottom. Click on the terminal window and type:

```
python main.py
```

This will run the Python file and all of the code within it. Once it finishes running, you should see "Hello world!" printed to the console. And that's how we will run the code from VSC!



Setting Up Our Pygame Project

Let's start our project by creating a folder to hold our project files. I've called mine "crossy rpg game" and placed it on the desktop but the name and location are totally up to you. Now open up VSC, select File -> Open, and then **select the folder you just created**. This will open the folder up in VSC which allows you to examine its content in the file explorer (top button on the far left side) and create new files. Create a new file by selecting File -> New and save it as **"main.py"** by selecting File -> Save As... This **automatically converts the file to a Python file** for writing Python code. Finally, make sure that you are using the latest version of Python.

We can access the Pygame library by importing it which we do by adding the statement:

```
import pygame
```

At the top of any Python file. Before using anything that is part of the pygame library, we must **initialize it** by adding this code:

```
pygame.init()
```

This performs any **necessary Pygame setup** so that we can access anything pygame related by calling pygame. and then the function/variable/object name. At the **end of your program** you should call...:

```
pygame.quit()  
quit()
```

... **to shutdown any Pygame functionality** and close the program properly. Any game code will go in between pygame.init() and pygame.quit()

Don't forget to save your changes every time!



Creating the Game Window

The first thing we should do in our program is create a game window. We can do so by calling the **pygame.display.set_mode() function** and passing in **width and height** as a tuple like this:

```
width = 800
height = 800
game_window = pygame.display.set_mode((width, height))
```

This creates a window that is **800 pixels wide and 800 pixels high** and stores the window in the **game_window variable**. In general, it is better practice to create variables and pass them in; this makes the program easier to modify and maintain than passing in literal values every time.

We can then color in the game window by calling the .fill function and passing in an RGB code. **An RGB code is a tuple of three numbers that represent the Red, Green, and Blue** color values of a color. Each ranges from 0-255 inclusive with 0 being the least amount of that color and 255 being the most. Different combinations of these colors can produce any color imaginable. For now, we will fill the window with a pure white like this:

```
white_colour = (255, 255, 255)
game_window.fill(white_colour)
```

The final step is to update the display by calling:

```
pygame.display.update()
```

This will render any graphics updates so **call this after drawing and filling in all colors**. We are now ready to run the program for the first time! We can do so by opening up the terminal window in VSC and running this code:

```
pythonw main.py
```

The **w at the end of python** ensures that the window can receive events such as key presses. As long as Pygame was installed correctly, you should see a couple of lines welcoming you to Pygame and telling you the version and whatever we put in the print statement. All this does for us right now is run the code, perhaps display a window for a split second, then finish running.

```
(base) Nimishs-MacBook-Pro:crossy rpg game nimishnarang$ pythonw main.py
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
(base) Nimishs-MacBook-Pro:crossy rpg game nimishnarang$ █
```

The problem is that **we are only running the display code once and then quitting**. To get it to persist, we need to put it in a Run Loop or Game Loop.



At the heart of **every game runs a massive loop**. This loop is started when the game runs and only breaks when something happens to stop the game from running such as when the user quits the game or does something to make the game crash. As this loop runs, it performs 3-4 main tasks: **handling input, updating game state, and rendering graphics** (the 4th task is **checking for endgame conditions** but that is often done during the updating game state). The loop continues to perform these tasks in order while the game runs to ensure that all possible events are handled before any updates and the state and graphics are updated as quickly as possible to provide a smooth experience.

Creating the Game Loop for Our Project

We can simulate this with a simple while loop which checks for some endgame variable or simply runs while True. We will implement the second case as it saves having to create an extra variable but that means it's up to us to manually break out of the while loop when the quit conditions are met. For now, we can just put the actual graphics rendering part in the loop so our main.py file will look like this:

```
import pygame

pygame.init()

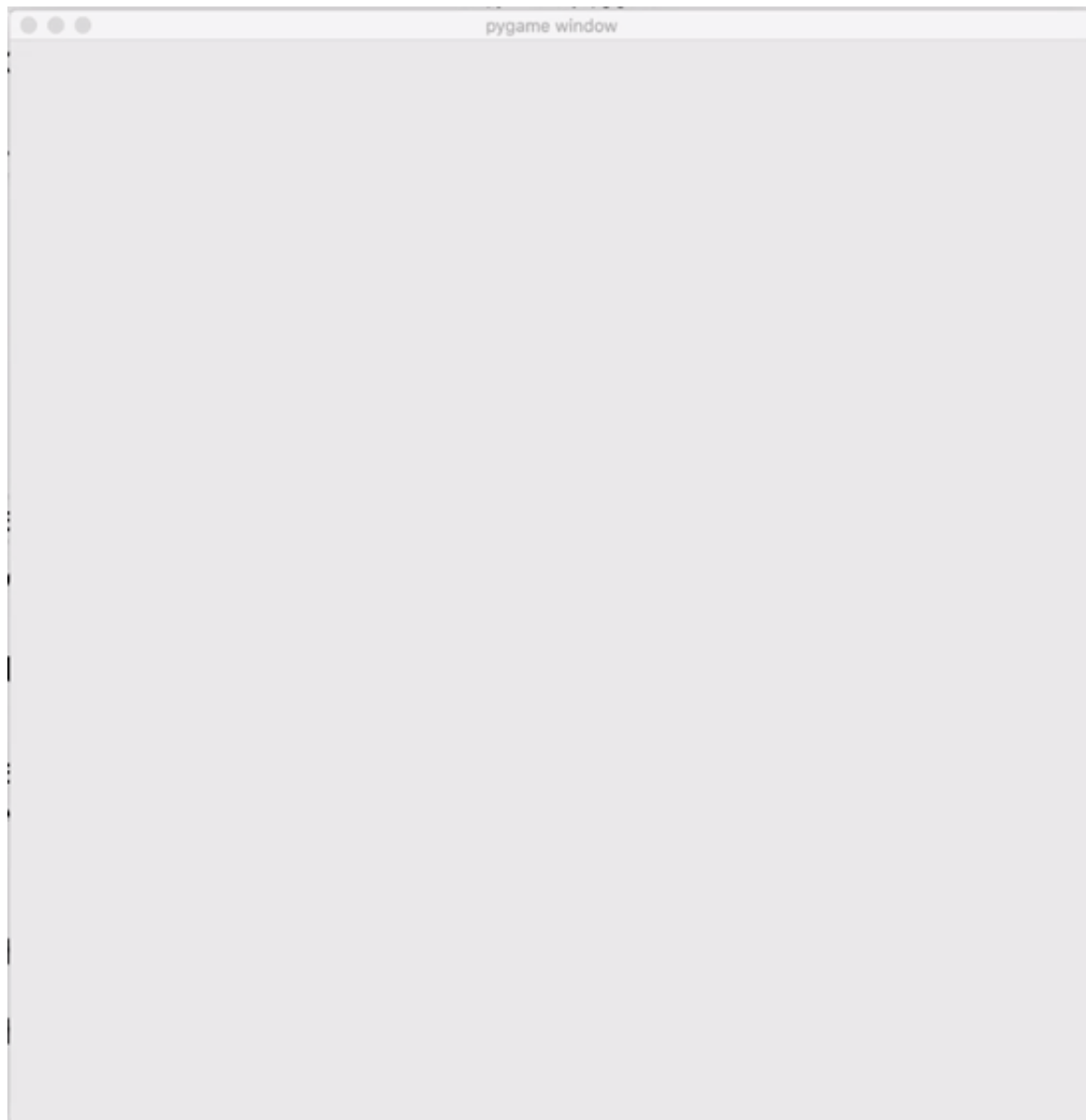
width = 800
height = 800
white_colour = (255, 255, 255)

game_window = pygame.display.set_mode((width, height))

while True:
    game_window.fill(white_colour)
    pygame.display.update()

pygame.quit()
quit()
```

Note how we create the variables outside of the loop. This is because **if we put them in the loop, we would create a new one with each loop iteration**, not what we want. If we run the program now (run `pythonw main.py` in the terminal window), we will see that the game window stays open but kind of gets stuck and doesn't allow us to close it. We have to **manually kill the program by running control+c** in the Terminal window. We can solve this problem by adding a clock and a quit event handler.





This while loop for our Game Loop will continue to run infinitely, so it's up to us to ensure that eventually it will break when we go to run the program. To ensure that the loop will break when the user quits the game, we can listen for the quit event.

Creating a Quit Event

With each loop iteration, we want to **poll all occurring events and if any of them are a quit event, we can break:**

```
while True:
    events = pygame.event.get()
    for event in events:
        if event.type == pygame.QUIT:
            # break here
```

In the loop, we get all current events and store them in events. We loop through these events and **listen for a pygame.QUIT event** (triggered when the user closes the game window or otherwise quits the game). However, if we put a break statement in that if statement, that would only break out of the for in loop, not the while loop as **break and continue apply only to the inner loop**. Instead, we can put this in a function and return from it:

```
def run_game_loop():
    while True:
        events = pygame.event.get()
        for event in events:
            if event.type == pygame.QUIT:
                return

        game_window.fill(white_colour)
        pygame.display.update()

run_game_loop()
```

By returning, **we break out of all forms of control flow and break out of the function**. By filling the window every time, we effectively wipe clean any images or anything else that was drawn in the previous loop iteration.

Adding a Clock

The final thing to add to the run loop here is a Clock variable. **The Clock is used to determine how many times per second we want to run the game loop**. The higher the clock number, the smoother the experience because we update the state and the graphics more often. However, a higher clock number means more processing so we typically don't exceed 60. This is **similar to FPS** that you might experience in a game. We can create a Clock variable in the run_game_loop() function and tick it (set the number of updates per second at the end of the loop like this:

```
clock = pygame.time.Clock()

def run_game_loop():
    while True:
        events = pygame.event.get()
```

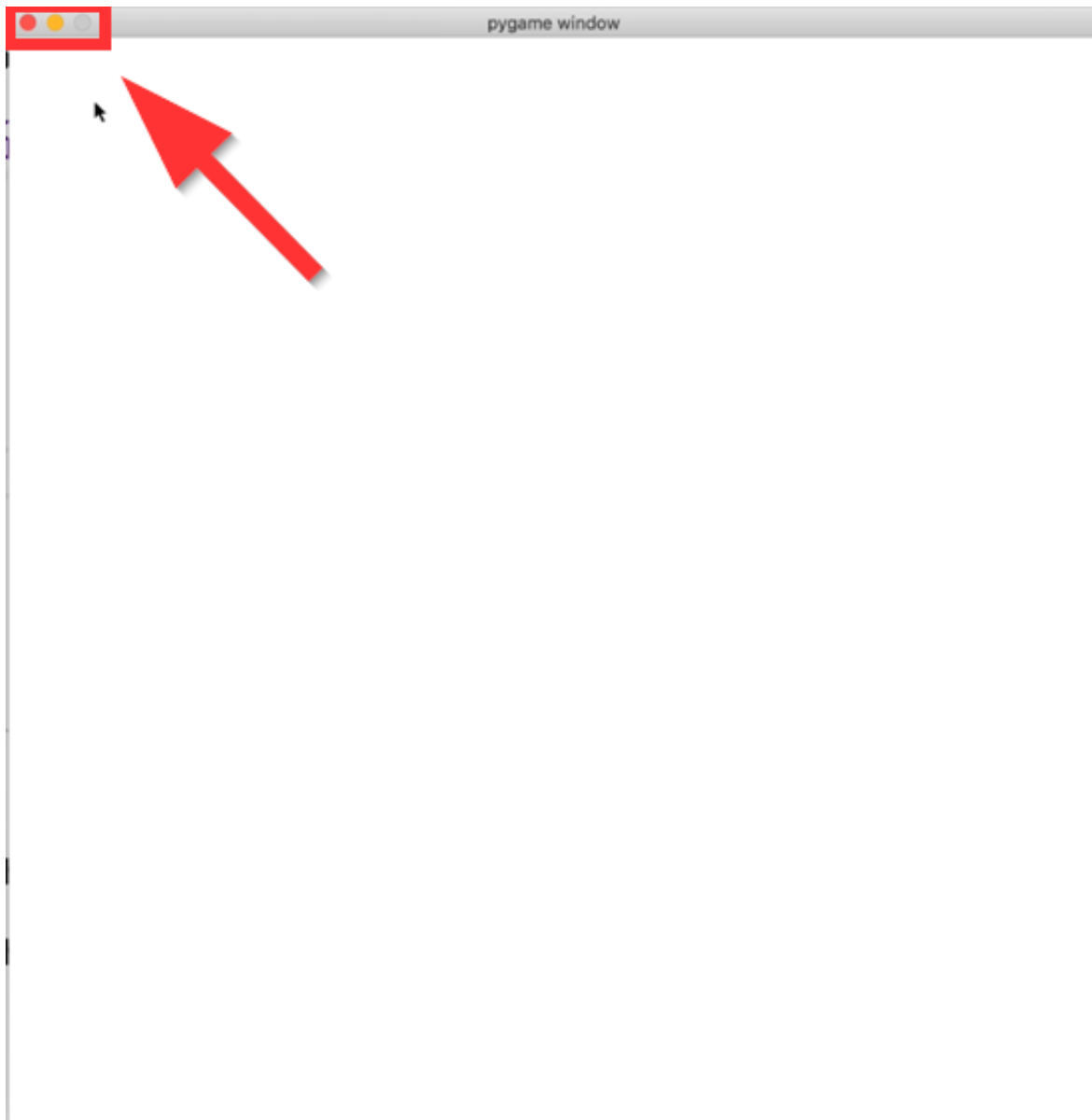


```
for event in events:
    if event.type == pygame.QUIT:
        return

game_window.fill(white_colour)
pygame.display.update()

clock.tick(60)
```

Now if we go to run the program, we should see that the window has been properly filled in with white and we can quit the program simply by x-ing out of it, triggering the quit event.





We can treat our game window like a sketch pad; we **draw any images to it through a process calling “blitting”**. We first import an image and create a variable to hold the image. We then blit that image onto the game screen.

Adding Images to the Project

We first have to add some images to the project. You should have an “assets” folder with your source code files that contains 4 images (background.png, enemy.png, player.png, and treasure.png). Copy that folder with the images **into your project folder** (so your project folder should contain a main.py file and the assets folder). Alternatively, you can provide your own images. Now that we have access to some image files, we can **load them into the project by providing a path to where they are stored**. For example, we load the background like this:

```
background = pygame.image.load("assets/background.png")
```

This looks into the assets folder and looks for “background.png” (every time we go into a new folder in a file path we use a “/”. If this doesn’t work, try a “\”). Put this code **just below the code to create the game_window**. Now that we have the image loaded, we blit it to the screen at the assigned x and y values like this:

```
game_window.blit(background, (0, 0))
```

Where background is the loaded image and **(0, 0) are (x, y) values with x=0 being far left and y=0 being the top** of the window. We put the blit function call between `game_window.fill(white_colour)` and `pygame.display.update()`. This ensures that the background is **drawn after the game window** has been wiped. If we run the code, we should see a persistent white window with an image taking up about a quarter of the screen. The next step is to scale the image up to fill the whole screen. We can do so by calling the **pygame.transform.scale function and passing in the image to scale and the (final_width, final_height)** of the image:

```
background = pygame.transform.scale(background, (width, height))
```

This is the same size as the game window and we want the background to take up the entire window.

Drawing Other Items

If we want to draw other items, **we can blit them on top of the background by simply blitting them after we blit the background**. Try loading in the treasure image in the same way as the background, scaling it up to 50×50, and blitting it at (375, 50). After you do this, our main file should look like this:

```
import pygame

pygame.init()

width = 800
height = 800
white_colour = (255, 255, 255)
```




```
game_window = pygame.display.set_mode((width, height))

clock = pygame.time.Clock()

background = pygame.image.load("assets/background.png")
background = pygame.transform.scale(background, (800, 800))

treasure = pygame.image.load("assets/treasure.png")
treasure = pygame.transform.scale(treasure, (50, 50))

def run_game_loop():
    while True:
        events = pygame.event.get()
        for event in events:
            if event.type == pygame.QUIT:
                return

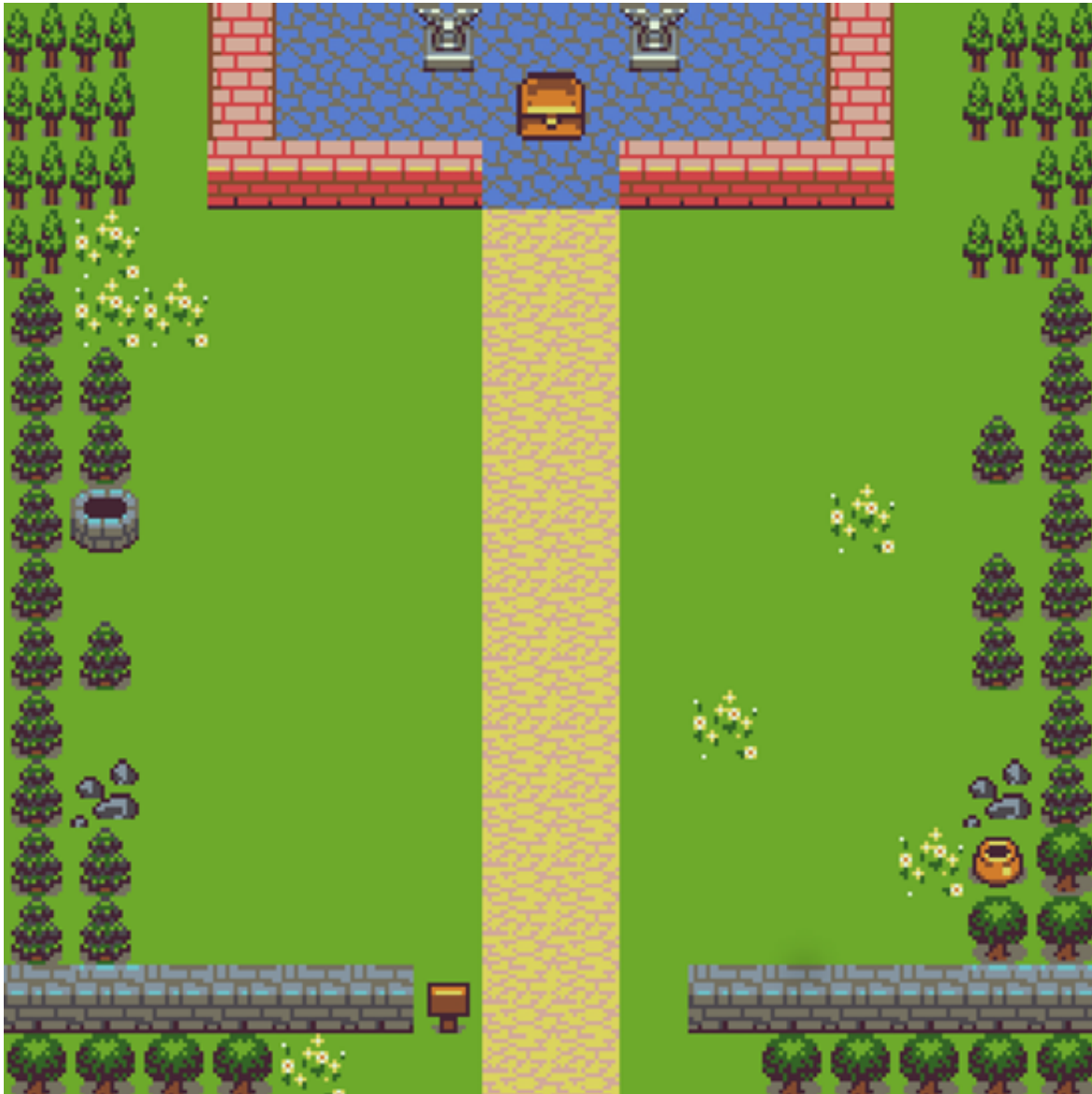
        game_window.fill(white_colour)
        game_window.blit(background, (0, 0))
        game_window.blit(treasure, (375, 50))
        pygame.display.update()

        clock.tick(60)

run_game_loop()

pygame.quit()
quit()
```

If you run the program, you should see this:





At this point, our `run_game_loop()` function is getting too cluttered and we shouldn't be putting all of this code in `main.py` anyway. Let's create a separate Game class in a new file.

Creating Game.py

Create a new file called "game.py" and import pygame. Create a class called Game and move our `run_game_loop()` function into it. A lot of the setup code (creating the screen and the images) can be moved **out of the `run_game_loop()` function and into the initializer and variables** like the clock, `game_screen`, `background`, and `treasure` can become properties of the **Game class**. We should create width and height variables as we will be using them at several points throughout the class.

In the **`run_game_loop()` function**, be sure to **reference `self.game_screen`, `self.background`, etc.** as these are now properties of the Game class. We should also separate out drawing functionality as we have several lines to deal with that now. Put the drawing code in a separate function and just call that function from within the game loop. By the end, the `game.py` file will look like this:

```
import pygame

class Game:

    def __init__(self):
        self.width = 800
        self.height = 800
        self.white_colour = (255, 255, 255)

        self.game_window = pygame.display.set_mode((self.width, self.height))

        background = pygame.image.load("assets/background.png")
        self.background = pygame.transform.scale(background, (self.width, self.height))

        treasure = pygame.image.load("assets/treasure.png")
        self.treasure = pygame.transform.scale(treasure, (50, 50))

        self.clock = pygame.time.Clock()

    def draw_objects(self):
        self.game_window.fill(self.white_colour)

        self.game_window.blit(self.background, (0, 0))
        self.game_window.blit(self.treasure, (375, 50))

        pygame.display.update()

    def run_game_loop(self):

        while True:
            events = pygame.event.get()
            for event in events:
                if event.type == pygame.QUIT:
                    return

            self.draw_objects()
```



```
self.clock.tick(60)
```

Accessing the Game.py

In order to run any of this, we need to create an instance of Game in main.py and call the `run_game_loop()` function. To access, Game, we have to **import the class** with this statement:

```
from game import Game
```

This reads: from the file game.py, import the class Game. After **pygame.init()**, we can create an instance of Game and run the loop with this code:

```
game = Game()  
game.run_game_loop()
```

This should **result in exactly the same result as before** when we go to run the game but our project is much better structured now and is ready for the next steps. If you run the file you should see the exact same results as we haven't changed anything; we have just moved the code and improved the project structure.



Most of the objects in our game will have mostly the **same properties: x, y, width, height, and image**. Therefore, it makes sense to create a custom class to hold these properties. That way, the code is more maintainable and cleaner.

Creating the GameObject Class

Create a new file called `gameObject.py` and a new class called `GameObject`. We will also need to import `pygame` as we need access to the image load and scale functionality that it provides. This will just have an **initializer that takes in x, y, width, height, and image_path values**. We can then use the `image_path` to load an image and the width and height passed in to scale the image and set it as a property:

```
import pygame

class GameObject:

    def __init__(self, x, y, width, height, image_path):
        image = pygame.image.load(image_path)
        self.image = pygame.transform.scale(image, (width,height))

        self.x = x
        self.y = y
        self.width = width
        self.height = height
```

This way, we can **access a GameObject's x, y, width, and height to place it and detect boundaries and the image property to blit the object's image**.

Make Treasure a GameObject

Note that treasure and background have exactly the same functionality for now so these can both become `GameObjects`. In `game.py`, we can import the `GameObject` class:

```
from gameObject import GameObject
```

And replace the background and treasure loading code to this:

```
def __init__(self):
    self.width = 800
    self.height = 800
    self.white_colour = (255, 255, 255)

    self.game_window = pygame.display.set_mode((self.width,self.height))

    self.background = GameObject(0, 0, self.width, self.height, 'assets/background.png')
    self.treasure = GameObject(375, 50, 50, 50, 'assets/treasure.png')

    self.clock = pygame.time.Clock()
```



However, now background and treasure are **GameObjects, not just images**, so we need to replace the two lines where we blit the background and the treasure to these:

```
self.game_window.blit(self.background.image, (self.background.x, self.background.y))  
self.game_window.blit(self.treasure.image, (self.treasure.x, self.treasure.y))
```

Note how we are fetching the properties of image, x, and y assigned during initialization rather than using hardcoded values. Although this may look like more effort, **this is a much better practice** as if we can change the properties of background or treasure to immediately see an effect rather than changing the values in potentially multiple places.



Our player character is going to share the same basic properties as a `GameObject` (x, y, width, height, and image) but will need some extra stuff. We want to create a new class for the player character (`Player`) and it makes sense to **subclass the `GameObject` to save ourselves from repeating code.**

Creating the Player Class

Like with `GameObject`, we should create a new file to hold this class but we don't need to import `pygame` as `GameObject` contains all of the `pygame`-specific functionality that we might need. We do, however, have to import the `GameObject` class:

```
from gameObject import GameObject
```

```
class Player(GameObject):  
    # definition here
```

We will set a new property: **speed in the initializer** and will later use this to move our player character around the screen. Our initializer will look very similar to the `GameObject` initializer but **will take in an additional speed property**. We want to call the superclass initializer to save from repeating code like so:

```
def __init__(self, x, y, width, height, image_path, speed):  
    super().__init__(x, y, width, height, image_path)  
  
    self.speed = speed
```

For now, let's provide a base movement implementation and discuss some potential challenges. The player will **move only up or down so we only need to change the y position**. One way to do this might be to take in a direction (-1 for up, 1 for down, and 0 for standing still) and use it to modify whether speed is positive or negative. We can then change the y value by speed like this:

```
def move(self, direction, max_height):  
    self.y += (direction * self.speed)
```

Adding the Player to the Game

We will improve this implementation as we go but for now, let's add a `Player` to the `Game` class. First, import `Player`:

```
from player import Player
```

And **create a `Player` variable in the initializer** centered in the x direction and near the bottom in the y direction:

```
self.player = Player(375, 700, 50, 50, 'assets/player.png', 10)
```




We can then display the player by blitting them just after the treasure:

```
self.game_window.blit(self.player.image, (self.player.x, self.player.y))
```



In order to move the PlayerCharacter, we need to detect **whether the arrow keys are being pressed and then call the move() function**. To do this, we need to add more test cases to our event handler.

Listening for Keydown Events

First, we need to listen for **keydown events**. If detected, we determine which **key was pressed** and adjust a **player_direction variable** to -1 if the up arrow was pressed and 1 if the down arrow was pressed. Add a variable called player_direction to the top of the run_game_loop() function:

```
player_direction = 0
```

And this check in the event handler:

```
elif event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        player_direction = -1
    elif event.key == pygame.K_DOWN:
        player_direction = 1
```

This first detects whether the events list contains a keydown event (when a key is pressed down) of any key. We then check to see if that key was the up or down key and update player_direction. It may seem counterintuitive to have a negative direction if the up key is pressed but **the screen's 0,0 is top left, not bottom left** so a decrease in y position corresponds with moving upwards, not an increase.

If you run the program now, you will see that the player moves up and down but continues to do so even after we lift our fingers off of the keys. Also, the character can go offscreen at the top or the bottom.



Adding Keyup Events

We can fix the first issue simply by adding some additional test cases to handle keyup events as well. We check to see if there is a **keyup event** and if so, **whether or not the keyup is the up or down arrow key**. If it is, we can change the direction to 0 to stop the player from moving. We can add this additional clause after the KEYDOWN check:

```
elif event.type == pygame.KEYUP:
    if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
        player_direction = 0
```



The final event handler should look like this:

```
events = pygame.event.get()
for event in events:
    if event.type == pygame.QUIT:
        return
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
            player_direction = -1
        elif event.key == pygame.K_DOWN:
            player_direction = 1
    elif event.type == pygame.KEYUP:
        if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
            player_direction = 0
```

Right after the event handler logic and before the drawing logic, we can add a call to move the player like this:

```
self.player_character.move(player_direction)
```

This will ensure any movement takes place before redrawing the screen. Go ahead and run the game to see your player character moving up and down when you press the up and down keys and stopping when you lift up on the keys:



While we can move the player, the problem remains that the user can still go offscreen.

Adding Screen Boundaries

To fix this, we need to pass a boundary into the `move()` function and not update the y position if the player is at the top or the bottom. We start by adding in a `max_height` parameter:

```
def move(self, direction, max_height):
```

We can then use this to determine if we're at the bottom of the screen and return if we are (same with the top):

```
def move(self, direction, max_height):
    if (self.y >= max_height - self.height) or (self.y == 0):
        return
    self.y += (direction * self.speed)
```

This returns before updating `self.y` if the player is at the bottom (`max_height - self.height`) or if they are at the top (`self.y == 0`). The reason we do `max_height - self.height` is because **the top of the player is at (0, 0) and we want to stop movement when the bottom of the player** (`self.y + self.height`) reaches the bottom.

Now when we call the function, we pass in the **game_window height**:

```
self.player_character.move(player_direction, self.height)
```

However, there is now another issue in which the player gets stuck at the top or the bottom. This is because we are **halting all movement when the player reaches the top or bottom**. Instead, we should only stop upward movement at the top and downward movement at the bottom like this:

```
def move(self, direction, max_height):
    if (self.y >= max_height - self.height and direction > 0) or (self.y == 0 and direction < 0):
        return
    self.y += (direction * self.speed)
```

Now when we run the code, we should see the player moving and stopping properly.





Now that we have the player moving around, let's add an Enemy class and an enemy to the game. Similar to the Player, **this will be a subclass of the GameObject class.**

Creating the Enemy Class

Like before, Create a file (this time called enemy.py), import GameObject, and add the class declaration:

```
from gameObject import GameObject

class Enemy(GameObject):
    # implementation here
```

We will add an **initializer** very similar to the Player class:

```
def __init__(self, x, y, width, height, image_path, speed):
    super().__init__(x, y, width, height, image_path)

    self.speed = speed
```

We need a move function but the implementation will be different from that of the Player. The move function will **only move the enemy left and right**, not up and down. Also, the movement will be **constant with the direction switch handled automatically** so no need to take in a direction parameter. We need a max_width instead of a max_height but the logic is the same. Instead of passing in the direction, we will automatically change the speed to -speed when the enemy reaches the far right and +speed when the enemy reaches the far left. Here is an implementation:

```
def move(self, max_width):
    if self.x <= 0:
        self.speed = abs(self.speed)
    elif self.x >= max_width - self.width:
        self.speed = -self.speed

    self.x += self.speed
```

Basically, if the **x position is at the far left (self.x <= 0)**, we make sure the speed value is **positive so the enemy starts moving to the right** and vice versa, turning the speed negative and moving the enemy left if they reach the far right. Once the speed has been set, we can just change the x position by that amount. Again, **we don't want the right-hand side of the enemy to go past the right edge** as self.x measures from the left side of the enemy.

Adding the Enemy to the Game

Now we want to add an enemy in our Game, update enemy movement with each while loop iteration, and draw the enemy. Start with the import in game.py:

```
from enemy import Enemy
```




And for now, add an enemy in the initializer:

```
self.enemy = Enemy(0, 600, 50, 50, 'assets/enemy.png', 10)
```

Add the enemy movement just under the player movement:

```
self.enemy.move(self.width)
```

And the blitting just under the player's blitting:

```
self.game_window.blit(self.enemy.image, (self.enemy.x, self.enemy.y))
```

If you run the game, you should see an enemy near the bottom of the screen moving from side to side:





At this point, none of the objects are colliding with each other but we should be detecting collision between the user and the enemy or the user and the treasure. As such, we will implement the collision detection mechanism in the Game class so that we can recognize a collision between any 2 objects.

Our collision detection will basically detect for overlap in one of the top or bottom edges and one of left or right edges. If just the left-right edges are overlapping but not the top-bottom edges, then the objects are above or below one another but not touching. If just the top-bottom edges are overlapping but not the left-right edges, then the objects are beside one another but not touching.

Adding Collision Detection - Height

We will create a function called `detect_collision()` which will take in two objects which we can use to check for overlap. We will first detect the y position overlap (but you can start with x if you want). We can do so with this code:

```
def detect_collision(self, object_1, object_2):
    if object_1.y > (object_2.y + object_2.height):
        return False
    elif (object_1.y + object_1.height) < object_2.y:
        return False
```

`object_1.y` refers to the top edge of the first object whereas `(object_2.y + object_2.height)` refers to the bottom edge of the object_2. **If object_1.y is greater than this**, it means that the object_1 is below the object_2. **The same logic is applied in the elif check but testing the bottom edge** of the object_2 against the top edge of the object_1. If either of these are the case, the objects could not possibly be touching and so we return False in both cases.

Adding Collision Detection - Width

We now want to perform the same logic but with the x edges:

```
if object_1.x > (object_2.x + object_2.width):
    return False
elif (object_1.x + object_1.width) < object_2.x:
    return False
```

The first check **returns False if there is no overlap between the left side of object_1 and the right side of the object_2** whereas the second check **returns False if there is no overlap between the right side of the object_1 and the left side of the object_2**. At the end of this function we **return True because if we haven't returned False by this point**, there must be overlap between the y and the x positions of the two bodies, therefore, there must be a collision. The final function looks like this:

```
def detect_collision(self, object_1, object_2):
    if object_1.y > (object_2.y + object_2.height):
        return False
    elif (object_1.y + object_1.height) < object_2.y:
        return False
```



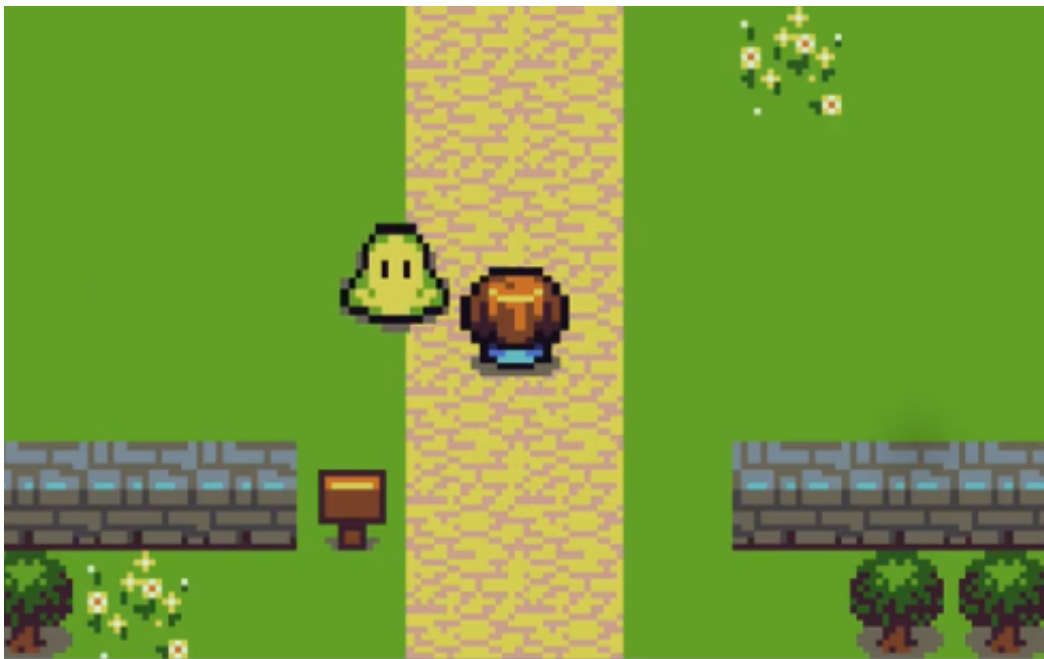
```
if object_1.x > (object_2.x + object_2.width):  
    return False  
elif (object_1.x + object_1.width) < object_2.x:  
    return False  
  
return True
```

Checking the Player & Treasure Collision

Now we will add the collision checks between the player and the treasure and the player and the enemy. For now, we will **return meaning we will quit the game** if there is a collision between the Player and the Treasure or between the Player and the Enemy. We do so just after we draw everything:

```
if self.detect_collision(player, enemy):  
    return  
elif self.detect_collision(player, treasure):  
    return
```

If you run the game, you should see that as soon as the player comes into contact with either the treasure or the enemy, the game shuts down and the window closes. Note that the **collision detection is not pixel perfect** due to the fact that the player and enemies have square boundaries but non-square images.



Having only one enemy in the game is boring so let's put a few enemies in **a list and use loops to update their movement, drawing, and collision detection**.

Adding Multiple Enemies

Start by replacing the single `self.enemy` with:

```
self.enemies = [  
    Enemy(0, 600, 50, 50, 'assets/enemy.png', 10),  
    Enemy(750, 400, 50, 50, 'assets/enemy.png', 10),  
    Enemy(0, 200, 50, 50, 'assets/enemy.png', 10),  
]
```

This ensures that we have a few enemies **spread out from top to bottom**. Also, note how the middle enemy starts at the far right instead of the left. Now when we go to draw the enemies, we need a loop. Replace the `draw self.enemy` line with this code:

```
for enemy in self.enemies:  
    self.game_window.blit(enemy.image, (enemy.x, enemy.y))
```

This **loops through each of the enemies** in the list and blits each one at the appropriate location. We need to apply the same logic to enemy movement, using a loop to move each enemy. We may as well put that in a separate function also:

```
def move_objects(self, player_direction):  
    self.player.move(player_direction, self.height)  
    for enemy in self.enemies:  
        enemy.move(self.width)
```

And call it from within the game loop just before drawing the objects. Note how we need to pass in the `player_direction` parameter:

```
self.move_objects(player_direction)
```

Updating Collisions

Finally, **loop through each enemy to detect collisions** along with the treasure-player collision. This also belongs in its own function that will return `True` if there is any collision and `False` otherwise:

```
def check_if_collided(self):  
    for enemy in self.enemies:  
        if self.detect_collision(self.player, enemy):  
            return True  
        if self.detect_collision(self.player, self.treasure):  
            return True  
    return False
```



Perform this check at the end of the game loop, just before the `clock.tick()` call:

```
if self.check_if_collided():  
    return
```

You should now see multiple enemies appear in the game window.





We want to execute two similar sets of logic when we collide with an enemy or with the treasure; we want to change the current level (affects the speed and number of enemies) and we want to reset the map. The big difference is that when we collide with an enemy, we **reset the level back to 1.0 and the number of enemies back to 1** whereas when we collide with the treasure, we **increase the level by 0.5 and possibly add more enemies**.

Resetting the Game

To start, we will create a Game property called level that starts at 1.0 so set this in the initializer of Game:

```
self.level = 1.0
```

Next, we want a **function to reset the map according to the current level**. This will involve creating lists of new enemies with the updated speed and resetting the player. We could do something like this to incorporate more enemies at the higher levels:

```
def reset_map(self):
    self.player = Player(375, 700, 50, 50, 'assets/player.png', 10)

    speed = 5 + (self.level * 5)

    if self.level >= 4.0:
        self.enemies = [
            Enemy(0, 600, 50, 50, 'assets/enemy.png', speed),
            Enemy(750, 400, 50, 50, 'assets/enemy.png', speed),
            Enemy(0, 200, 50, 50, 'assets/enemy.png', speed),
        ]
    elif self.level >= 2.0:
        self.enemies = [
            Enemy(0, 600, 50, 50, 'assets/enemy.png', speed),
            Enemy(750, 400, 50, 50, 'assets/enemy.png', speed),
        ]
    else:
        self.enemies = [
            Enemy(0, 600, 50, 50, 'assets/enemy.png', speed),
        ]
```

We start by **resetting the player character** to ensure they go back to their starting position. Next, we **set the speed according to the current level**. This only affects the enemies upon initialization so no need to make this a property of Game. Finally, we **create a list of either 1, 2, or 3 enemies** depending on the level.

Code Refactoring

As some of our setup is now occurring in the reset_map() function, we will replace the enemy and player creation code in __init__() with a call to reset_map() to get this result:

```
def __init__(self):
    self.width = 800
    self.height = 800
```



```
self.white_colour = (255, 255, 255)

self.game_window = pygame.display.set_mode((self.width,self.height))

self.clock = pygame.time.Clock()

self.background = GameObject(0, 0, self.width, self.height, 'assets/background.png')
self.treasure = GameObject(375, 50, 50, 50, 'assets/treasure.png')

self.level = 1.0

self.reset_map()
```

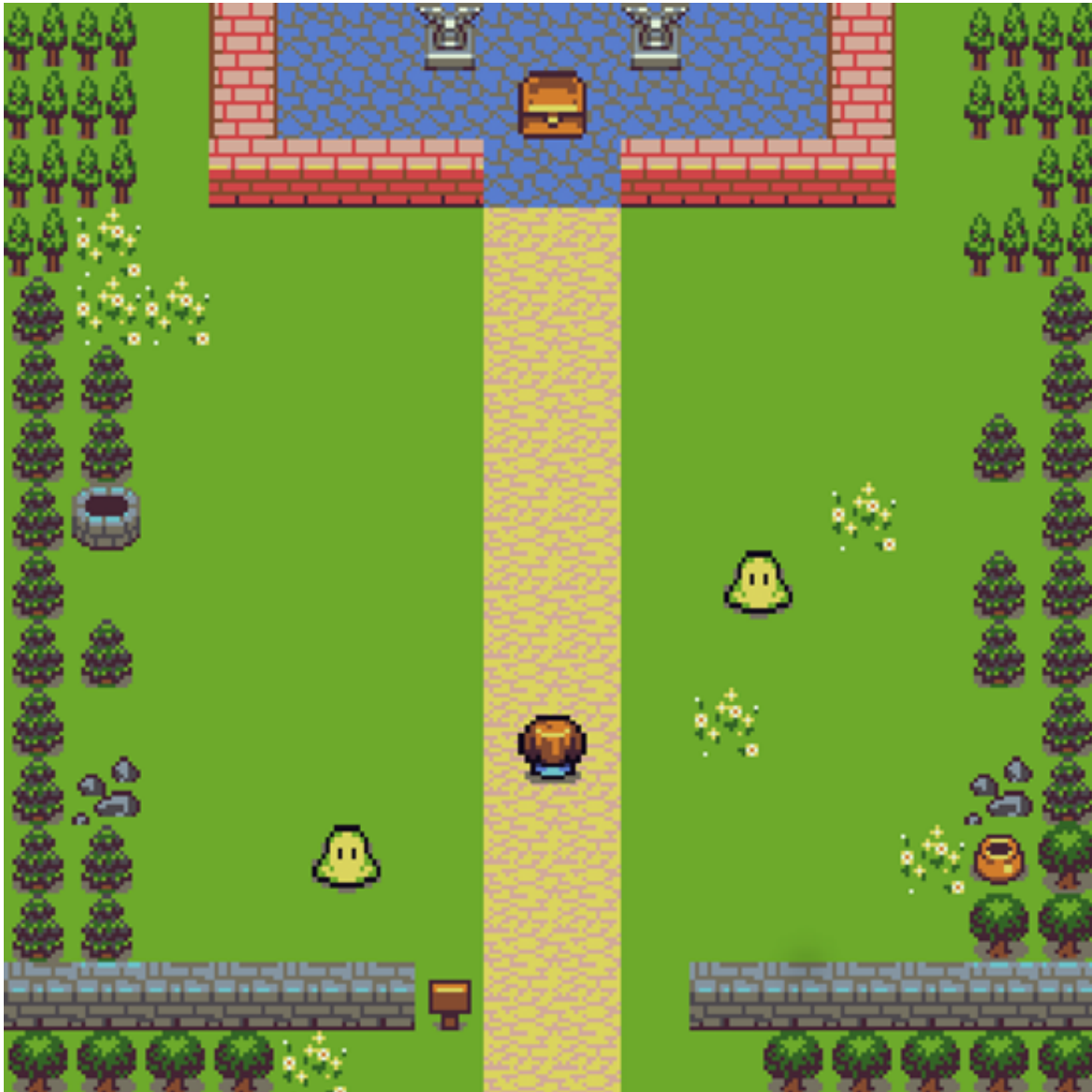
Our `check_if_collided()` function needs to update the level appropriately. If there is a collision with any of the enemies, we want to reset the level to 1.0 before returning. If there is a collision between the player and the treasure, we want to increase the level by 0.5 (a full 1.0 may increase the levels and speed too quickly). Our final implementation will look like this:

```
def check_if_collided(self):
    for enemy in self.enemies:
        if self.detect_collision(self.player, enemy):
            self.level = 1.0
            return True
        if self.detect_collision(self.player, self.treasure):
            self.level += 0.5
            return True
    return False
```

Finally, we need to reset the map in the game loop if there was any collision:

```
if self.check_if_collided():
    self.reset_map()
```

And that's it! We have now finished the game! Go ahead and test it out. You should see that as you win a few levels in a row, the enemies speed up and we add more into the mix. As soon as you touch an enemy however, everything resets and you go back to level 1. The final thing left to do is to summarize the course!





We're all done! Congratulations on making it to the end of the course! We installed Python, Anaconda, and Pygame, and used these to build a crossing-type game from scratch. By now you should feel proud that you have learned how to use Pygame, learned important project fundamentals and code practices, and have built a nifty little game.

From here, we recommend improving the game as a starting point. We know that what we have works, so it's just a matter of adding more features, changing up what we have to make it more functional, and incorporating new elements from Pygame. After that, you could start your own new projects such as new games or desktop applications.

Also, check out some other Python libraries to help you with your future tasks. There are tons of libraries out there for Python that can help perform all sorts of tasks from data science to machine learning to creating GUIs. We're super excited to see what sorts of projects you guys come up, and where your new Python careers will take you!

Thanks so much for taking this course with us, I hope that you learned something from it and enjoyed it and I look forward to hopefully seeing you in some future courses!