PYTHON

MIDDLE SCHOOL EDITION

ROLE PLAYING GAME

GAME DEVELOPMENT GUIDE

BY PIXELPAD

GAME DEVELOPMENT GUIDE

PIXELPAD

ROLE PLAYING GAME WORKBOOK

Second Edition

EDITORS

Jamie Chang
Ivo van der Marel
Arthur Teles
Rochelle Magnaye

DESIGNERS

Fernando Medrano
Kenneth Chui
Prateeba Perumal
Emily Chow

www.pixelpad.io
www.underthegui.com

CONTENTS

COURSE GOALS & LEARNING OUTCOMES

Students create a top down 2D Role Playing game, where they learn how to make a game similar to popular titles such as Zelda. Students create a world with various characters, items and challenges.

COURSE GOALS

- Students have an understanding of how to make large world games and have the ability to modify code to add new features to a game in PixelPAD
- Students finish with a full role playing game with challenges and a large world

LEARNING OUTCOMES

Computational Thinking and Algorithms

- Students have an understanding of how to manipulate variables algebraically and through computation on the PixelPAD platform
- Students have an understanding of commenting and how to comment their code on PixelPAD and Python
- Students are able to write readable and properly structured code
- Students understand the concept of a function f(x) in the context of the PixelPAD environment such as room_set("room"), collision_check(self, "obj")
- Students are able to create arrays in code and store or retrieve information from these arrays

Creativity

- Students are able to generate a full world with their own characters, who can interact and talk with the player
- Students can implement a narrative in the world they have created
- Students can create items to provide the player with extra abilities

Prototyping, Testing and Debugging

- Students are able to read the console, find errors and fix basic errors themselves

Construction

- Students will have the ability to make a full project in PixelPAD, with little support of an instructor

Communication & Collaboration

- Students will be able to read and understand other students code, even if different from theirs
- Students can reflect other students work and have other students reflect their work

SAMPLE PROJECT

The book's project can be found here:

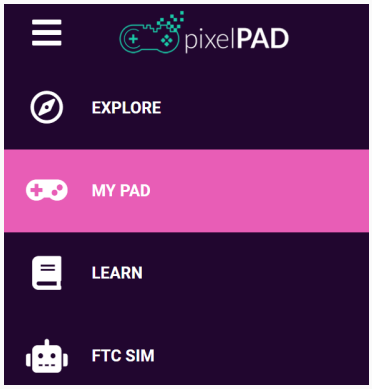
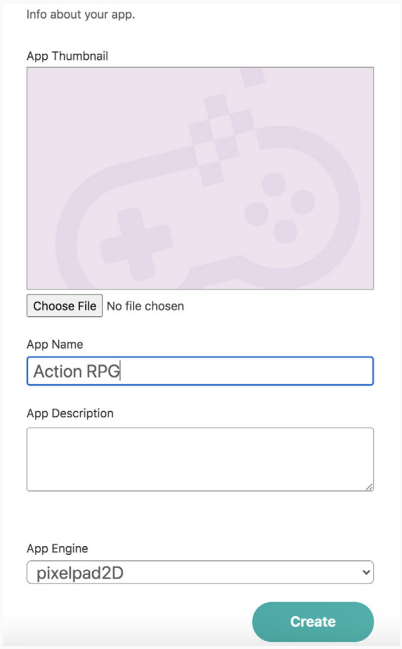
https://pixelpad.io/app/txspnzsmoud/?edit=1

01.
CHAPTER

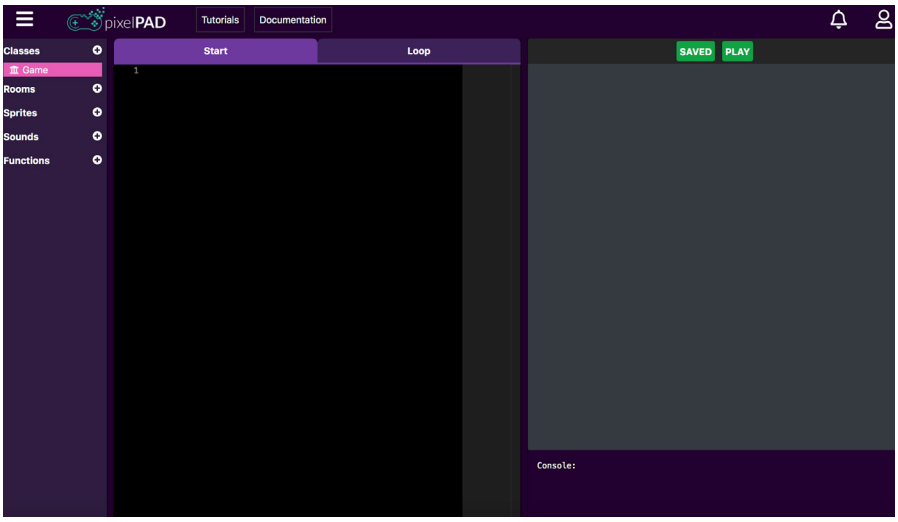
SETTING UP

Welcome to the Role Playing Game (RPG) course! We're going to create a top down RPG like this!

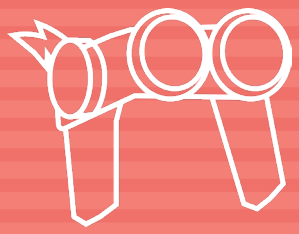
First, we will need to create a new project in pixelPAD. Once you've logged in you want to:

Click on the "MY PAD" button on the left, if you don't see it click on the hamburger icon (three lines) next to pixelPAD.	
Then click on the blue button "Create App".	
A window should pop-up, type in your App Name. Feel free to name it any way you like. We named it "Action RPG". **Make sure the App Engine is "pixelpad2D".** **Click "Create" when you're done.**	

Once you created your project, you should see the PixelPAD Development Environment page.



02.
CHAPTER



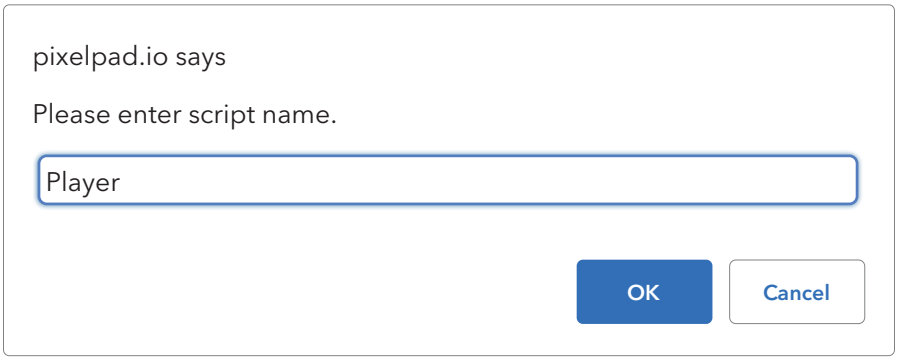
THE PLAYER

Our game right now is totally empty. Let's add the most important piece first: our player!

The first step is to create a Player class to be our player by clicking in the plus icon next to the classes menu in bold white

Then, a popup window will show up asking us to give the class a name. Name the class Player, as shown below and make sure to capitalize the "P" in Player. When you are done, click OK.

Capitalizing the first letter of a class name is important because it helps distinguish classes from regular variables (more on variables later).



You should now see the Player class you just created right below the Game class. The pink highlight means you have opened that class. In this image, the Player class is open.

Now, we will add some code that will make the Player appear on the screen!

Click on the Game Class, then click on the Start Tab of your editor.

Pay attention to the **pink** and **purple** tabs! These tell you where to type in your code.

Add the following **bolded code** below.

Game | Start

```
self.pixelhead = Player()
```

This line of code creates something called a **variable**.

A variable is used to store some value, this one we named pixelhead. With it, we are storing an object of type 'Player', using the command Player(). Feel free to give your player any name you like!

Notice how we've used the words classes and objects in this project so far. Don't be confused by them! A class is simply a blueprint that we use to construct/create the objects that we interact with in our game. For example, our Player class is the blueprint from which we create "actual" Player player objects such as pixelhead.

The **self** attributes the created object to the class this code is written in, in this case, the Game class.

In plain english, this line would look like this: "pixelhead" is my object from the class Player.

For in-depth explanation of the Start and Loop Tabs, head to the glossary section that explains The Game Loop!

SAVE + PLAY

Save and Play your game to see the changes

Whenever you add content to your game it is a good idea to press save at the top right of your game window!

You should see a blue box that says "Empty Image".

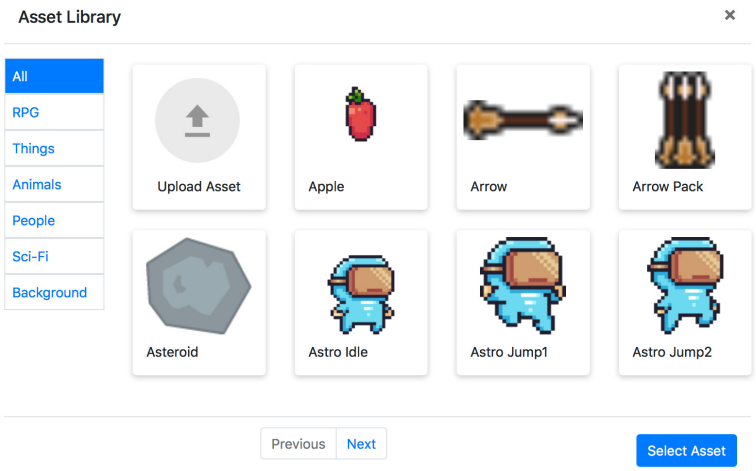
You may notice that your object shows up, but it says empty image. This is because we haven't assigned our object an image. In game development terms, an image is called a "sprite".

A sprite is a computer graphic that is moved on a screen or manipulated. Generally, sprites refer to 2D images that make up our game's art.

So for our Player object, we will need to assign a sprite.

To do this, first click on the plus icon to the right of the Sprites menu.

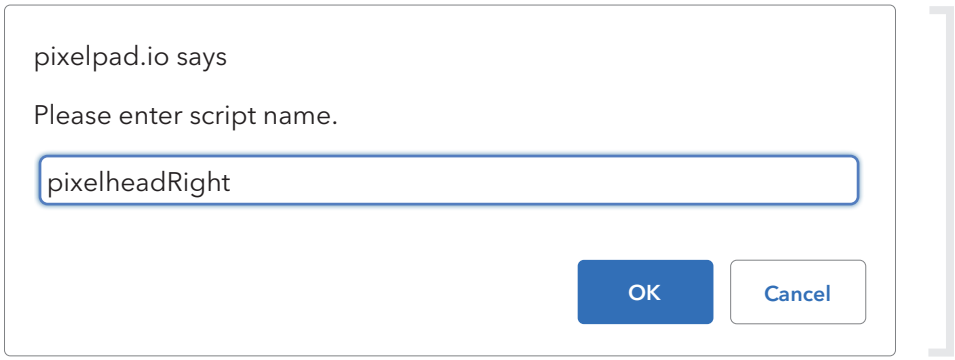
This will open up the PixelPad's Asset Library, which will allow you to either select an existing sprite or upload one from your computer!



NOTE: If you've decided to upload a sprite from your computer, make sure that it is transparent and that it has a .png extension. Do **NOT** choose an animation sprite such Pixelhead Idle or any sprite that looks like it's moving. Animations will **NOT** be covered in this project.

For our example, we will be using the Pixelhead Right sprite.

Now, a popup window will pop up asking you to give your sprite a name. Name your sprite "pixelheadRight" as shown below and click OK.



Click on the Player Class, then click on the Start Tab of your editor.

Player | Start

```
self.sprite = sprite("pixelheadRight.png")
```

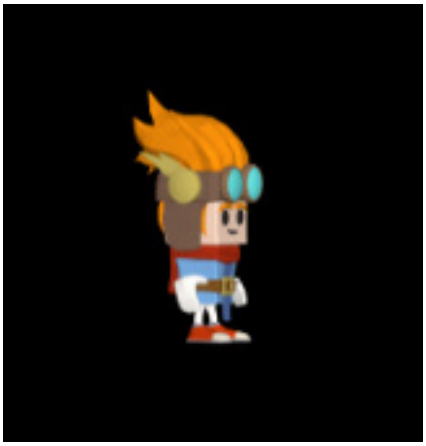
Here we are assigning a sprite to your pixelhead.

Notice again, the word self. We use it to call the object's sprite. That is, the self keyword allows us to refer to the sprite that belongs to the Player object.

SAVE + PLAY

Save and Play your game to see the changes

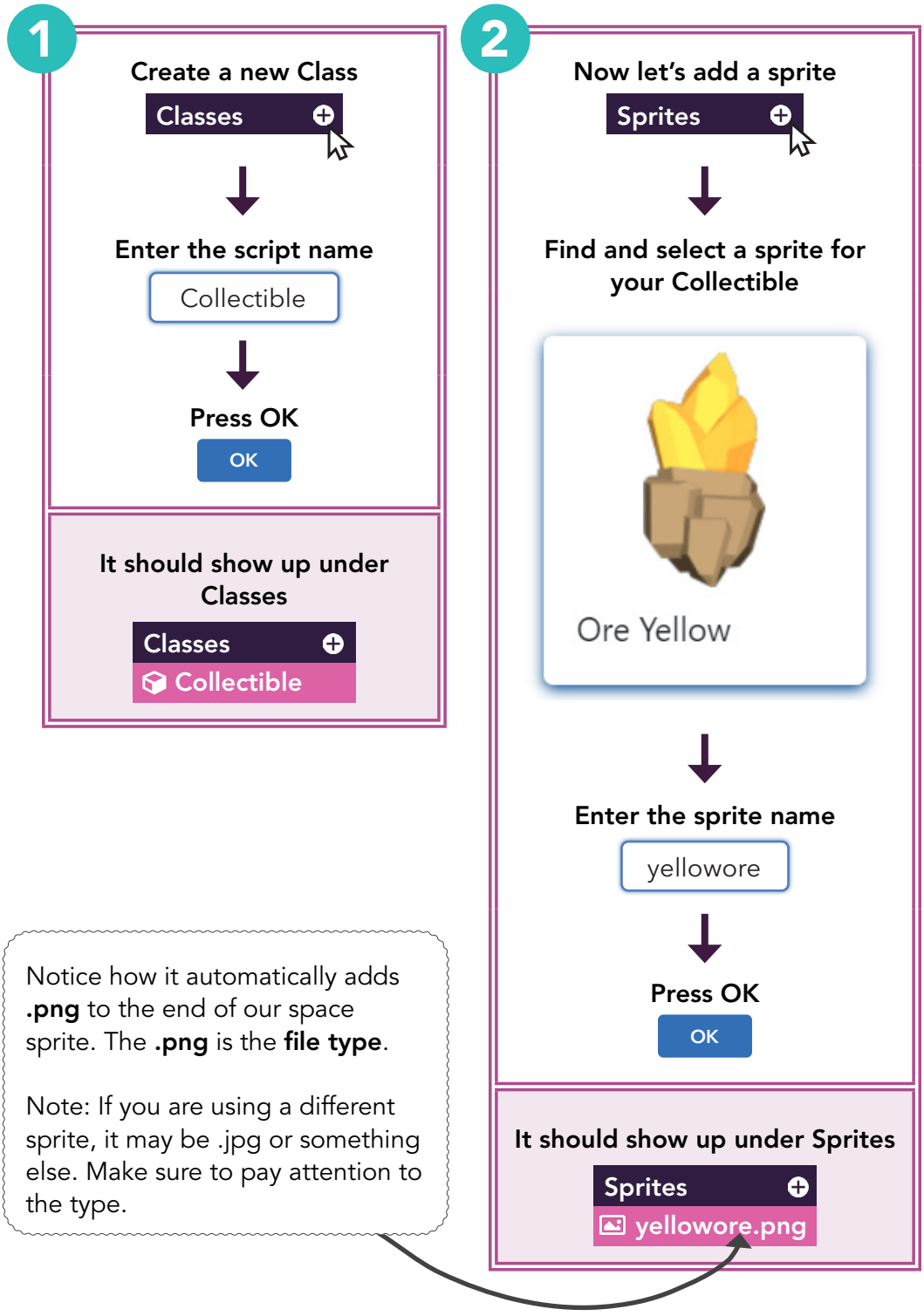
You should now notice that your player has a sprite associated with it!



PROPS

Let's create some props like ores to decorate the room (scene).

Like the Player, we also have to create a Class and get the Sprite from the asset library before coding it inside the Game class.



Notice how it automatically adds **.png** to the end of our space sprite. The **.png** is the **file type**.

Note: If you are using a different sprite, it may be .jpg or something else. Make sure to pay attention to the type.

After creating a class and selecting an asset, we are ready to add it to our game

Click on the Collectible Class, then click on the Start Tab of your editor.

Collectible | Start

```
self.sprite = sprite("yellowore.png")
```

Here we are simply assigning the ore sprite to the Collectible object.

SAVE + PLAY

Save and Play your game to see the changes

Now, we need to add this ore object to our Game class.

Click on the Game Class, then click on the Start Tab of your editor.

The **black bolded code** is what you need to add while the grey code is what you have already typed in before. This is to let you know where you need to type the code.

Game | Start

```
self.pixelhead = Player()

self.ore1 = Collectible()

self.ore2 = Collectible()
```

Here we are creating two instances of a Collectible in our game.

Notice we have "ore1" and "ore2". These are called variables. The variable acts as a name so that we can tell which ore is which.

SAVE + PLAY

Save and Play your game to see the changes

However, if you play the game, we only see one ore! Why do you think this happened?

PLACING ORES

We only see one ore because they are both standing in the same place! So let's make them stand in different places by editing their X and Y values. X is how far left or right the object is. Y is how far up or down. Let's move "ore1".

Game Start

```
self.pixelhead = Player()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

self.ore2 = Collectible()
```

Here we are directly manipulating the x and y position of ore1 and placing it at different positions in our game.

SAVE + PLAY

Save and Play your game to see the changes

You can do the same for any object, let's move ore2.

Game | Start

```
self.pixelhead = Player()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

self.ore2 = Collectible()
self.ore2.x = -250
self.ore2.y = -80
```

Here we are directly manipulating the x and y position of ore2 and placing it at different positions in our game.

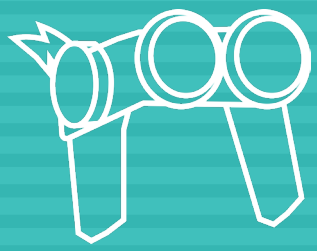
SAVE + PLAY

Save and Play your game to see the changes

Here we have added a few ores around your game. In programming, everything is grouped into a data type. For example, we used the number 100 earlier.

100 is in the format of the integer data type (often shortened to int). The x and y variables can only use int values.

03.
CHAPTER



PLAYER CONTROLS

We now have a bunch of objects placed in our game, but it's not much of a game if we can't interact with it at all.

Now, we're going to add movement to the player. To do this, we're going to be adding code to the Loop Tab, which runs again and again so behaviours that we code in the loop tab will happen again and again. For in depth knowledge about Start and Loop, head to "The Game Loop" section in the glossary.

Player | Loop

```
self.x = self.x + 1
```

This line refers to the player's current x position, and then sets it to itself + 1. This is simply updating the x position by adding 1 to it continuously.

The result is that the player will look like it is constantly moving to the right.

SAVE + PLAY

Save and Play your game to see the changes

You will notice that your player keeps moving to the right until it exits your game view!

This code is saying that whatever the x value was before, it adds one to it, and then does that again and again so fast it looks like it is moving smoothly. Kind of like a flipbook animation!

Now try changing the number!

What happens when it's a higher number? What if it's negative? What if you replace x with y? Test these out, you should be able to get the player to move in any direction.

As fun as it is to watch our player run off in every direction, we want to have control over when they move and in which direction. To do that, we are going to need an If Statement.

Notice how "self.x = self.x + 1" is **indented**? This is intentional. If you do not have an indent, you can press the **Tab** key on your keyboard to add it.

The crossed out code means you can remove it because we don't need it anymore.

Player Loop

```
self.x = self.x + 1
if key_is_pressed("arrowRight"):
    self.x = self.x + 1
```

This if statement checks if the right arrow key is pressed.

If it is, the player movement code from before is called, moving our player to the right like before.

Notice how the if statement is structured, there is an indent on the line below it. This is because we are creating an if statement and everything that is indented and below it is enclosed in it. That means that "self.x = self.x + 1" is enclosed inside "if key_is_pressed("arrowRight"):.

SAVE + PLAY

Save and Play your game to see the changes

Now the player should only move when you press down on the right arrow key. Try it out!

Player | Loop

```
if key_is_pressed("arrowRight"):
    self.x = self.x + 1

if key_is_pressed("arrowLeft"):
    self.x = self.x - 1

if key_is_pressed("arrowUp"):
    self.y = self.y + 1

if key_is_pressed("arrowDown"):
    self.y = self.y - 1
```

Like our first if statement, the other if statements here move our character in different directions.

The major differences between our first if statement and the rest are whether or not we add or subtract a number, and whether or not we are adding or subtracting to x or y.

Changing x will make us move horizontally, changing y will make us move vertically.

SAVE + PLAY

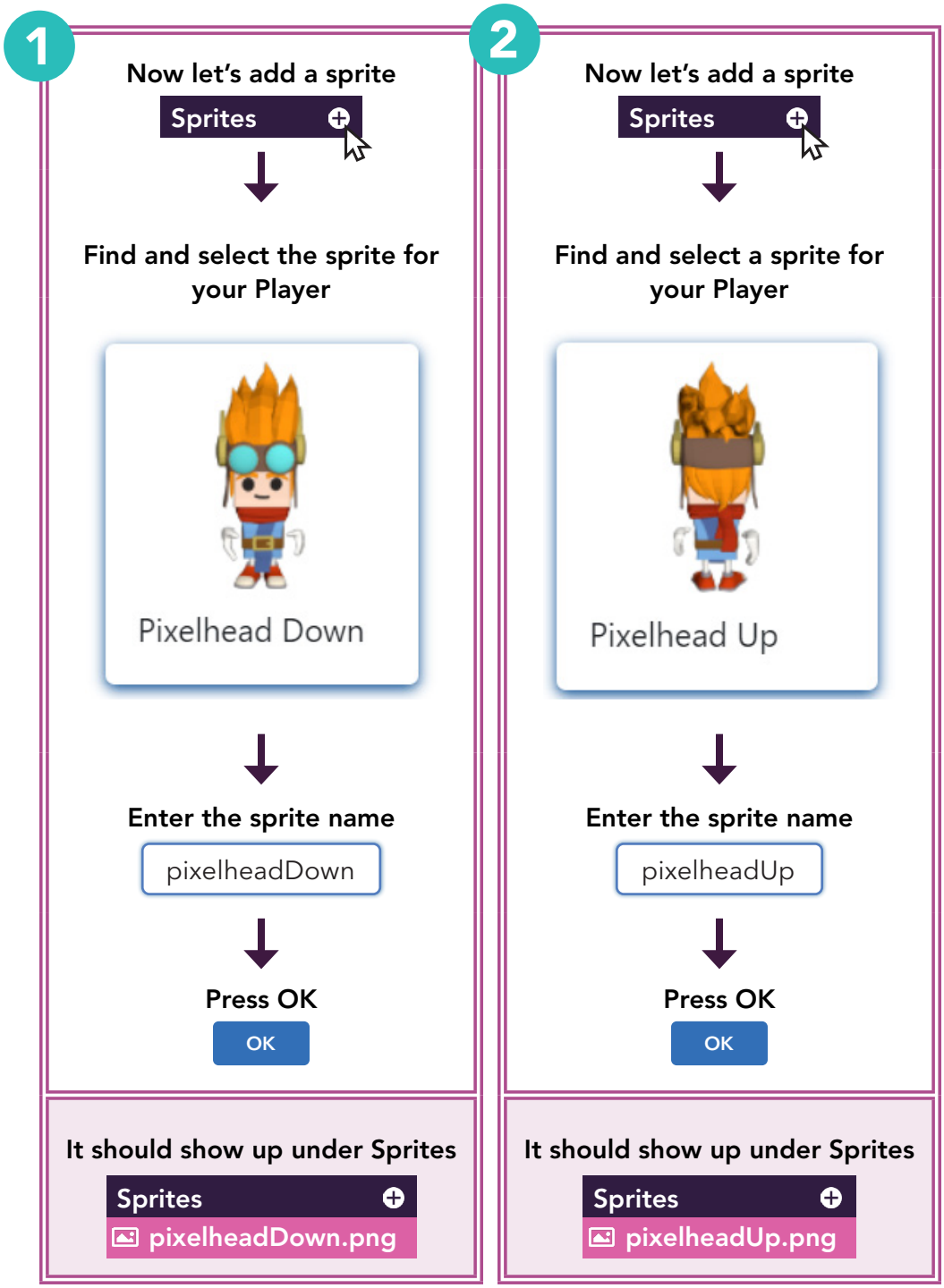
Your player character should now be able to move in every direction using the arrow keys! Test it out and you should be able to move around smoothly. You can also change the speed of your player by changing the number that you add or subtract by.

PLAYER DIRECTIONS

Our player is now able to move in all directions. However, notice how the direction they are facing does not match the direction they are moving in? Well, in this section, we will fix that!

To fix this, whenever the player moves in a direction, we'll change their sprite.

First, Let's open the sprite asset store. Notice how we have different sprites for our player: Pixelhead Down, Pixelhead Right, and Pixelhead Up. Since we have Pixelhead Right uploaded already, let's upload the rest!



Player | Loop

```
if key_is_pressed("arrowRight"):
    self.x = self.x + 1
    self.sprite = sprite("pixelheadRight.png")

if key_is_pressed("arrowLeft"):
    self.x = self.x - 1

if key_is_pressed("arrowUp"):
    self.y = self.y + 1
    self.sprite = sprite("pixelheadUp.png")

if key_is_pressed("arrowDown"):
    self.y = self.y - 1
    self.sprite = sprite("pixelheadDown.png")
```

Here, inside out if statements, we've attached the appropriate sprite to the Player so that it matches the direction the player is moving in. For example, in our first if statement, we checked if the user has pressed the right arrow key, if they do the player moves in the right direction and the sprite of the player should change so that they are facing right.

SAVE + PLAY

Notice how we don't change our sprite when we press the left arrow. That's because we don't have a pixelheadLeft sprite. But this is not a problem, we can easily flip the pixelheadRight sprite by setting their scaleX value to -1.

Every object has default scaleX and scaleY values set to 1. If we invert the scaleY value, we flip the object vertically, and if we invert the scaleX value, we flip it horizontally.

The **X** is upper case for **scaleX**. Make sure to pay attention to **Upper and Lower cases**. The code may not work if they are not typed correctly!

Player Loop

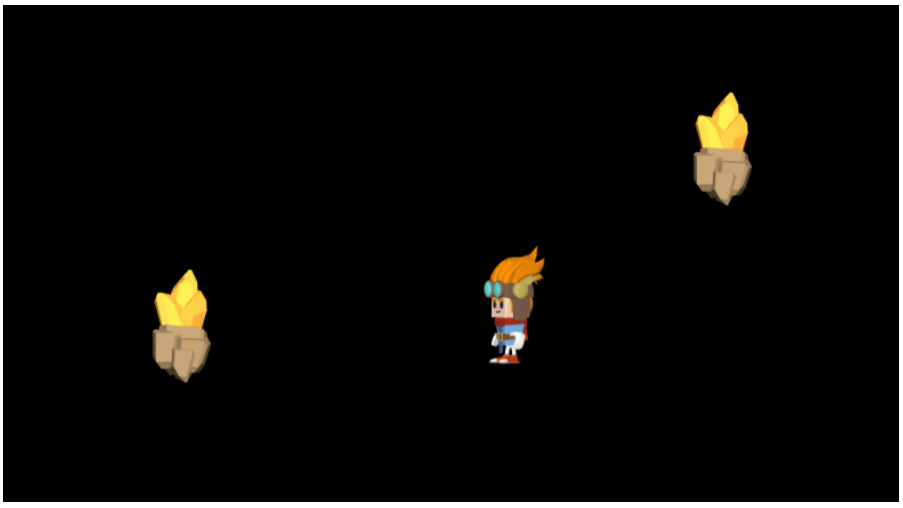
```
if key_is_pressed("arrowRight"):
    self.x = self.x + 1
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = 1

if key_is_pressed("arrowLeft"):
    self.x = self.x - 1
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = -1

if key_is_pressed("arrowUp"):
    self.y = self.y + 1
    self.sprite = sprite("pixelheadUp.png")

if key_is_pressed("arrowDown"):
    self.y = self.y - 1
    self.sprite = sprite("pixelheadDown.png")
```

We first make sure that our player's scaleX is 1 when moving to the right.

Next, we set the sprite pixelheadRight when moving to the left and flip the sprite image by setting the scaleX to -1.

SAVE + PLAY

Your player should now be moving in all directions!



When we move our player, we always add or subtract 1 from our x or y positions. That's how fast our player moves. We can also increase our player's speed by changing that to another number.

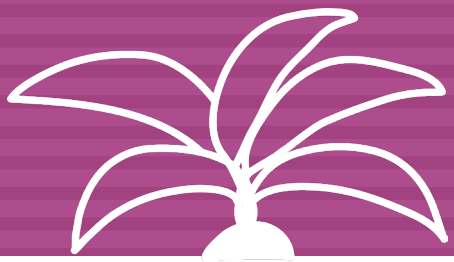
Player | Loop

```
if key_is_pressed("arrowRight"):
    self.x = self.x + 4
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = 1

if key_is_pressed("arrowLeft"):
    self.x = self.x - 4
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = -1

if key_is_pressed("arrowUp"):
    self.y = self.y + 4
    self.sprite = sprite("pixelheadUp.png")

if key_is_pressed("arrowDown"):
    self.y = self.y - 4
    self.sprite = sprite("pixelheadDown.png")
```

Here, we are increasing the player's speed by making it move 4 units instead of 1.

SAVE + PLAY

04.
CHAPTER



BACKGROUND

Adding a background into our game is pretty much the same as adding any object. For example, when we created our Player and Ores, we first created a class, attached a sprite and wrote some code to create these objects. Creating a background is pretty much the same.

1

Create a new Class

Classes ⊕

Enter the script name

Background

Press OK

OK

It should show up under Classes

Classes ⊕

Background

2

Now let's add a sprite

Sprites ⊕

Find and select a sprite for your Background

BG Forest

Enter the sprite name

forest

Press OK

OK

It should show up under Sprites

Sprites ⊕

forest.png

Notice how it automatically adds **.png** to the end of our space sprite. The **.png** is the **file type**.

Note: If you are using a different sprite, it may be .jpg or something else. Make sure to pay attention to the type.

Background Start

```
self.sprite = sprite("forest.png")
```

Here, we attached the sprite forest.png to our Background class.

Now, let's make our background appear on the screen!

Note: The "..." is to indicate there is more code below/above, you do NOT add the dots.

Game Start

```
self.forest = Background()

self.pixelhead = Player()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

...
```

Here, we've created an object of type Background that we named field.

Make sure that you write this code at the very beginning of your Game start tab. If you don't, the background will overlap the other objects.

SAVE + PLAY

COLLISIONS - PICKING UP ORES

Right now, our player just passes right through our ores without anything happening. We want our player to be able to collect them.

To do this, we're going to need to see if the player "touches" the ore. In other words, we will check if a "collision" happens between the player and a ore using a function called get_collision(). Head to the collisions section of the glossary for an in-depth explanation on collisions and get_collision()!

Player | Loop

```
if key_is_pressed("arrowRight"):
    self.x = self.x + 1
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = 1

if key_is_pressed("arrowLeft"):
    self.x = self.x - 1
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = -1

if key_is_pressed("arrowUp"):
    self.y = self.y + 1
    self.sprite = sprite("pixelheadUp.png")

if key_is_pressed("arrowDown"):
    self.y = self.y - 1
    self.sprite = sprite("pixelheadDown.png")

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
```

Here, we create a variable that stores information on whether or not a collision between the player (self) and any Collectible object has occurred.

Then, inside an if statement, we check if a collision has happened. If it did, we simply destroy (remove) the Collectible object that the player has collided with.

SAVE + PLAY

You should now be able to collide with ores, and when you do, they are destroyed!

Still, however, we don't get any value from getting any ores. So, let's make the player earn some money every time they collect an ore!

Player Start

```
self.sprite = sprite("pixelheadRight.png")

self.money = 0
```

Here, we simply create a variable that stores the amount of money that the player has and set it to 0 as a starting point.

Player Loop

```
...

if key_is_pressed("arrowDown"):
    self.y = self.y - 1
    self.sprite = sprite("pixelheadDown.png")

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
```

Here, we increase the money variable by 1 every time the player touches a ore.

Great! Our player should now be able to collect ores and increase their money each time. However, we don't really have any way of visually seeing if things are working as we expect. How do we know if our money variable is actually increasing by 1 every time the player collides with it?

Luckily, Python has what's called a print function that allows us (programmers) to display text to the console.

Here's how we can use the print function to see if our game is working as we want.

Player | Loop

```
...

if key_is_pressed("arrowDown"):
    self.y = self.y - 1
    self.sprite = sprite("pixelheadDown.png")

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
    print("Money: " + str(self.money))
```

Here, when a collision between the player and a Collectible object happens, we use the print function to display a message to the console that tells us the amount of ores that the player has collected so far.

Notice how the message that we wanted to see in the console is wrapped inside single quotations and brackets. For example, if we want to display the message "hello there!" using the print function, we would write:

print("Hello there!")

Next, you'll notice another function called str(). This simply converts any number that is wrapped inside its parenthesis, such as self.money to a string (text). This is important because you are not allowed to mix numbers and text (strings) without doing some conversion.

Lastly, the process of "gluing" text (strings) and numbers together using the "+" operator is called string concatenation.

SAVE + PLAY

Printing messages to the console is useful and helps you locate bugs (errors) in your code.

05.
CHAPTER

ENEMIES

Let's add some dangerous creatures to the room! Let's start by adding an enemy to create some challenge for our player!

We're going to add some enemies that our Player should avoid touching.

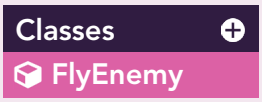
1

Create a new Class

Enter the script name

FlyEnemy

It should show up under Classes



2

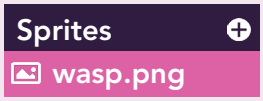
Now let's add a sprite

Find and select a sprite for your FlyEnemy

Enter the sprite name

wasp

It should show up under Sprites



FlyEnemy | Start

```
self.sprite = sprite("wasp.png")
```

Here, we attach a wasp sprite to our FlyEnemy.

Game Start

```
self.forest = Background()

self.pixelhead = Player()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

self.ore2 = Collectible()
self.ore2.x = -250
self.ore2.y = -80

self.wasp = FlyEnemy()
self.wasp.x = 400
self.wasp.y = -200
```

Here, we've created a FlyEnemy object that we've named wasp and gave it an x and y value of 400 and -200 respectively.

SAVE + PLAY

Great! We now have an enemy in our room! Feel free to add more wasps to your scene.

Now, let's make our FlyEnemy move so that it's not just sitting there.

FlyEnemy | Loop

```
self.x -= 1
```

Here, we simply update our FlyEnemy's x position by 1.

Notice the "-=". Don't be confused with this notation. It's actually more of an abbreviation to how we have been updating variables up until now.

For example,

self.x -= 1 is the same as self.x = self.x - 1.

SAVE + PLAY

Feel free to make your FlyEnemy move in any direction.

Let's now make our FlyEnemy destroy our Player once they collide!

Player | Loop

```
...

if key_is_pressed("arrowDown"):
    self.y = self.y - 4
    self.sprite = sprite("pixelheadDown.png")

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
    print("Money: " + str(self.money))

waspHit = get_collision(self, "FlyEnemy")
if waspHit:
    destroy(self)
```

Here, we create a variable that stores information on whether or not a collision between the Player (self) and any FlyEnemy object has occurred.

Then, inside an if statement, we check if a collision has happened. If it did, we simply destroy (remove from the game) the Player.

SAVE + PLAY

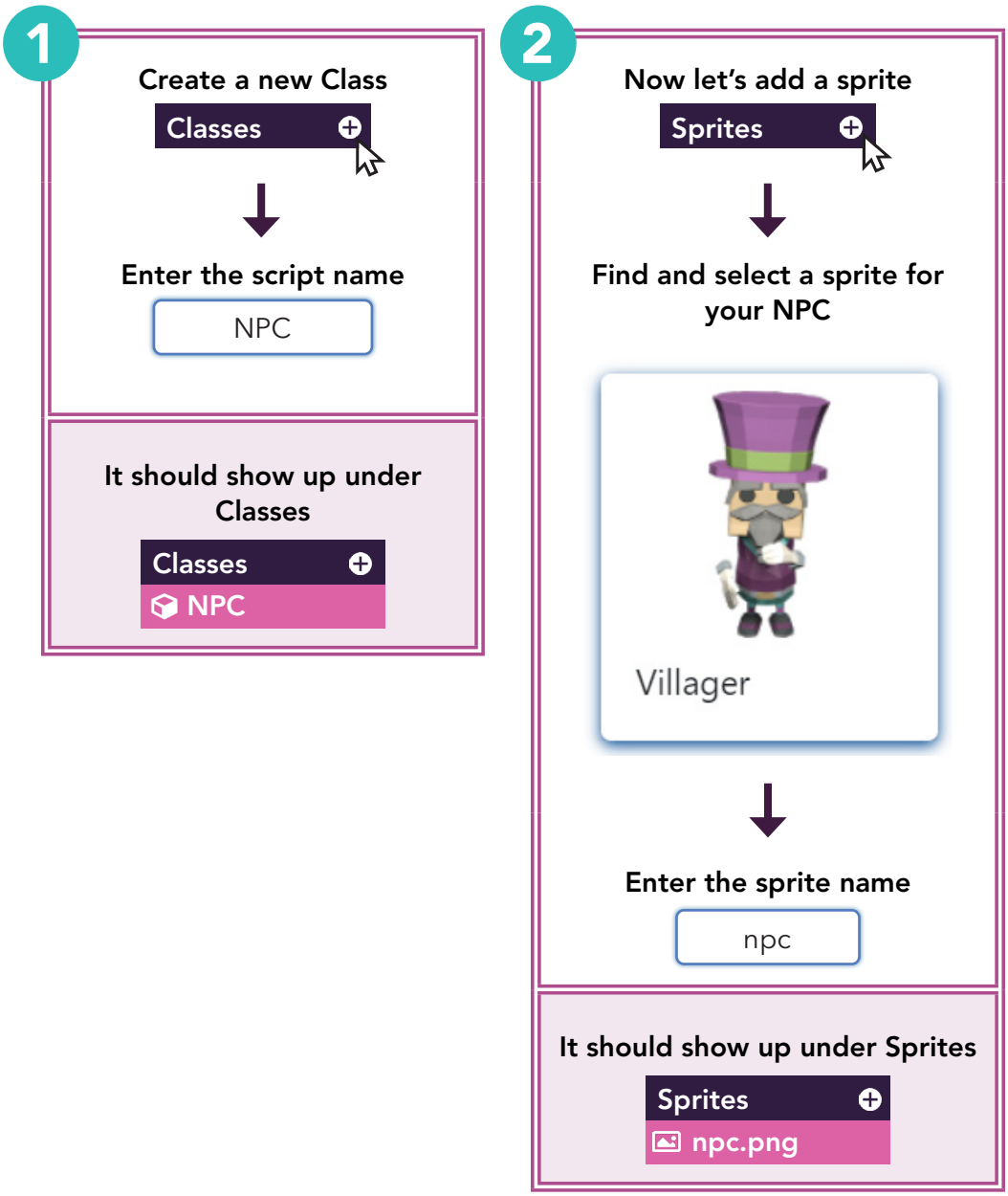
Later, we will add some health to the Player (our player) so that they don"t get destroyed in one hit.

06.
CHAPTER

NPC

NPCs are non-playable characters. In this game, our NPC will be a character that will help our Player by giving it a weapon to use.

Let's start by creating an NPC class and attaching a sprite to it from PixelPad's asset store.



NPC | Start

```
self.sprite = sprite("npc.png")
```

Here, we've attached the npc.png sprite to our NPC class.

Let's now add the npc to our game.

Game Start

```
self.forest = Background()

self.pixelhead = Player()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

self.ore2 = Collectible()
self.ore2.x = -250
self.ore2.y = -80

self.wasp = FlyEnemy()
self.wasp.x = 400
self.wasp.y = -200

self.mario = NPC()
self.mario.y = -250
```

Here, we've created an NPC object that is named mario and have given it a y value of -250. As we haven't assigned an x value, it remains 0.

SAVE + PLAY

Let's now add some code that will allow our Player to interact with the NPC.

Let's start by talking to the NPC!

NPC Loop

```
playerHit = get_collision(self, "Player")
if playerHit:
    if key_was_pressed("E"):
        print("It is dangerous out there. Take this sword!")
```

Here, we've created a variable called playerHit and we use it to store the collision information that the get_collision function gives us.

Then, inside another if statement we check if the "e" key was pressed using the function key_was_pressed().

Lastly, if the above two conditions are true (i.e. both events happen), we display a message to the console using a print statement.

SAVE + PLAY

Great! Now, every time we're near the NPC mario and press the "e" key, he can communicate with us.

Let's now make our NPC character do more than just talk to us! We'll now make them give us a sword as well!

The first thing we'll need is a variable that keeps track of whether or not the Player has a sword. We'll add this variable to our player's start tab.

Player | Start

```
self.sprite = sprite("pixelheadRight.png")

self.money = 0

self.hasSword = False
```

Here, we simply create a boolean variable that keeps track of whether or not the Player has a sword.

We set it to False to tell the game that initially the Player does not possess a sword.

Later, when the Player has a sword, we will have to change it to True.

SAVE + PLAY

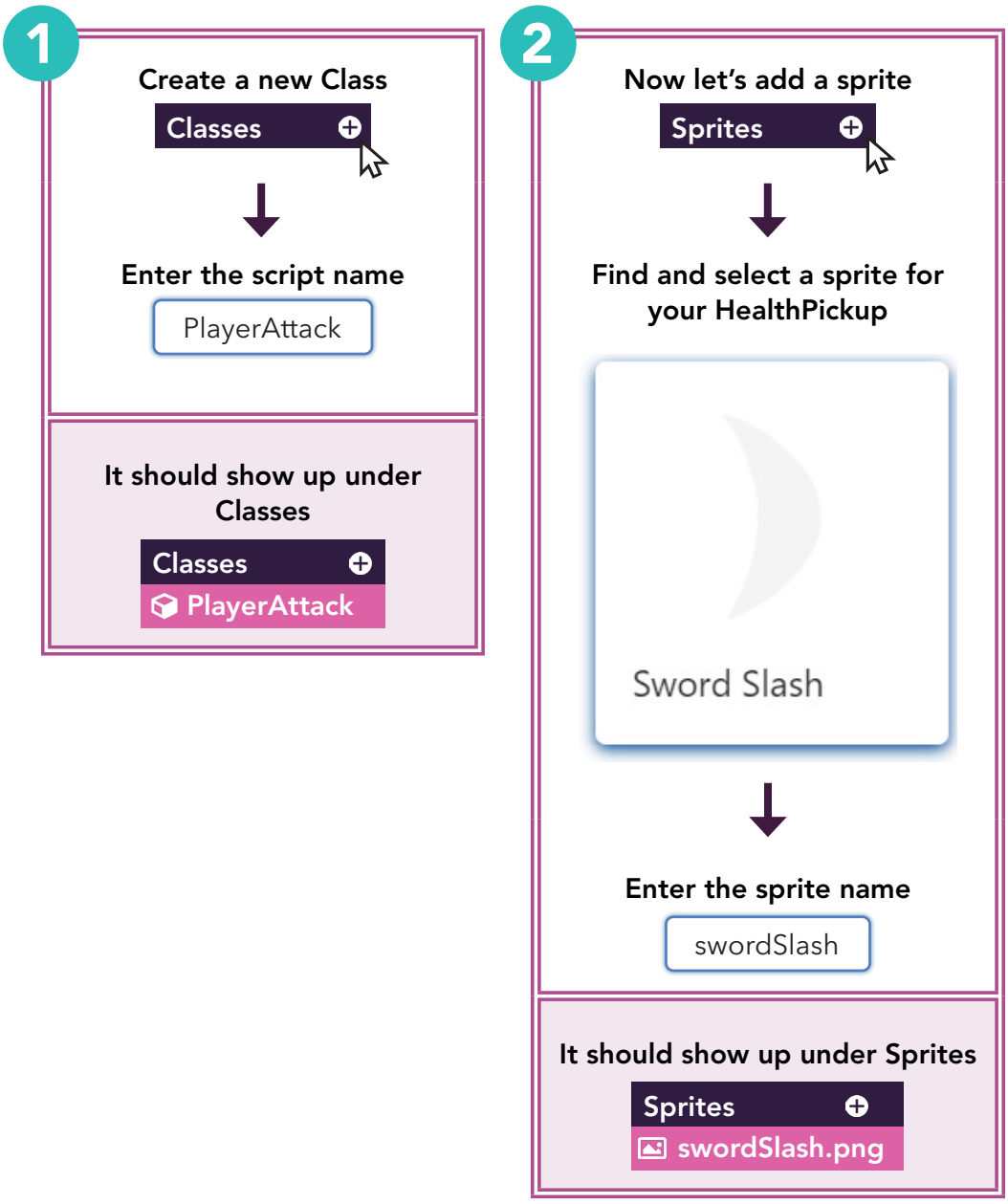
Let's update our NPC's loop tab so that when we touch an NPC object and the user presses "E", our hasSword variable is no longer set to False!

NPC | Loop

```
playerHit = get_collision(self, "Player")
if playerHit:
    if key_was_pressed("E"):
        print("It is dangerous out there. Take this
        sword!")
        playerHit.hasSword = True
```

Here, we've updated the player's hasSword variable to True to tell the game that the user has a sword.

The next thing we need to do so that the Player can perform an attack is to create a class for it and attach a sprite to it from the asset store.



PlayerAttack | Start

```
self.sprite = sprite("swordSlash.png")
```

Here, we attach the swordSlash.png sprite to our PlayerAttack class.

Let's now create an instance of our PlayerAttack inside the Player's loop tab.

Player | Loop

```
...

if key_is_pressed("arrowUp"):
    self.y = self.y + 4
    self.sprite = sprite("pixelheadUp.png")

if key_is_pressed("arrowDown"):
    self.y = self.y - 4
    self.sprite = sprite("pixelheadDown.png")

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
    print("Money: " + str(self.money))

waspHit = get_collision(self, "FlyEnemy")
if waspHit:
    destroy(self)

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
```

Here, whenever the player presses the "F" key and their self.hasSword variable is set to True, we create a PlayerAttack object that we named swordSlash.

SAVE + PLAY

Cool! Our Player can now unleash sword slashes whenever they interact with the mysteryMan NPC.

But before we move on, notice how after obtaining the ability to use sword slashes, no matter how many times you press the "F" key, we only see one sword slash. This is because every time we press the "F" key, we're creating a PlayerAttack object without changing its default position. Remember what the default position of any object? That's right, it's (0,0), meaning both the x and y values are 0s.

What we want to do is to create sword slashes and make them appear wherever the player is!

Player | Loop

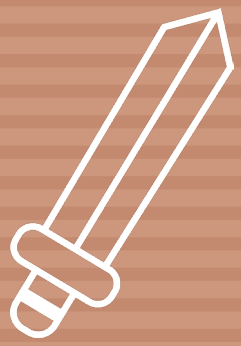
```
...

if key_is_pressed("arrowUp"):
    self.y = self.y + 4
    self.sprite = sprite("pixelheadUp.png")

if key_is_pressed("arrowDown"):
    self.y = self.y - 4
    self.sprite = sprite("pixelheadDown.png")

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
    print("Money: " + str(self.money))

waspHit = get_collision(self, "FlyEnemy")
if waspHit:
    destroy(self)

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
        swordSlash.x = self.x
        swordSlash.y = self.y
```

Here, whenever the player presses the "F" key and their self.hasSword variable is set to True, we create a PlayerAttack object that we named swordSlash.

SAVE + PLAY

Great! now our SwordSlashes spawn right where the Player is!

07.
CHAPTER



DIRECTIONAL ATTACKING

In the previous chapter, we learned how to interact with NPCs that can help our Player by providing them with a sword slash. In this chapter, we will make sure that these sword slashes face the direction that the Player is facing.

First, we'll need to create a variable inside the Player Class to keep track of the direction that the Player is facing.

Player | Start

```
self.sprite = sprite("pixelheadRight.png")

self.money = 0

self.hasSword = False

self.direction = "right"
```

Here, we create a variable called direction that stores the direction that the Player is facing.

We initially set it to"right", however, we will later update it accordingly inside the loop tab so that it matches the direction the Player is actually facing.

Now, let's update the direction variable inside the Player's loop tab.

Player | Loop

```
if key_is_pressed("arrowRight"):
    self.x = self.x + 4
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = 1
    self.direction = "right"
if key_is_pressed("arrowLeft"):
    self.x = self.x - 4
    self.sprite = sprite("pixelheadRight.png")
    self.scaleX = -1
    self.direction = "left"
if key_is_pressed("arrowUp"):
    self.y = self.y + 4
    self.sprite = sprite("pixelheadUp.png")
    self.direction = "up"
if key_is_pressed("arrowDown"):
    self.y = self.y - 4
    self.sprite = sprite("pixelheadDown.png")
    self.direction = "down"

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
    print("Money: " + str(self.money))

...
```

First, we look for the code that we've written earlier that changes the Player's sprite based on the key that the user has pressed.

Then, we add some code that updates our direction variable so that it matches the Player's direction sprite!

If we click Save and Play now, we won't really see any difference in our SwordSlashes. This is because we haven't really done anything to change their direction.

Next, we will use the Player's direction variable in order to rotate the SwordSlash accordingly!

Player | Loop

```
...

waspHit = get_collision(self, "FlyEnemy")
if waspHit:
    destroy(self)

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
        swordSlash.x = self.x
        swordSlash.y = self.y
        if self.direction == "up":
            swordSlash.angle = 90
        if self.direction == "down":
            swordSlash.angle = -90
        if self.direction == "left":
            swordSlash.angle = 180
```

Here, inside the if statement that checks if the user has pressed the f key, we write 3 more if statements to check the direction variable's value.

Next, we use the angle command to rotate the sword slash sprite accordingly. For example, if the direction variable has the value "up", we rotate it 90 degrees clockwise. If the variable's value is "down", we rotate the sprite 90 degrees anti-clockwise (-90 degrees), and finally, if the direction is left, we rotate it 180 degrees.

Notice how we only check for 3 directions and make only 3 rotations. This is because the sprite is originally facing right, so there is no need to check or rotate it right again!

SAVE + PLAY

Great! Our swordSlash PlayerAttack objects can now face the same direction that the Player is facing!

Notice, however, That the PlayerAttacks spawn right in the middle of our Player. What we want is for them to spawn some small distance away from them so that they don't look like they're touching the Player.

For example, if the player is looking to the left, the attack should be created at the player's position and offsetted a bit more to the left.

Player | Loop

```
...

waspHit = get_collision(self, "FlyEnemy")
if waspHit:
    destroy(self)

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
        swordSlash.x = self.x
        swordSlash.y = self.y
        if self.direction == "up":
            swordSlash.angle = 90
            swordSlash.y += 80
        if self.direction == "down":
            swordSlash.angle = -90
            swordSlash.y -= 80
        if self.direction == "left":
            swordSlash.angle = 180
            swordSlash.x -= 80
        if self.direction == "right":
            swordSlash.x += 80
```

Here, we offset the slash position depending on what direction the player is looking at.

We needed to add the extra if statement to check now for the right direction because we also need to offset the attack to the right if the player is looking in that direction.

SAVE + PLAY

Nice, we've now gotten a bit of range to attack our enemies. However, notice that our PlayerAttacks remain in the game forever! That is, they don't disappear at all!

Let's now make these PlayerAttacks disappear after some time has passed using a timer variable!

PlayerAttack | Start

```
self.sprite = sprite("swordSlash.png")

self.timer = 5
```

Here, we create a timer variable that will be used as a countdown for our PlayerAttack objects to disappear.

Now, we'll decrease this variable by 1 until it reaches 0, and when it does, we'll destroy it so that it no longer stays in the game!

PlayerAttack | Loop

```
self.timer -= 1

if self.timer <= 0:
    destroy(self)
```

Here, decrease the timer variable by 1. Then, we check to see if it drops below 0. If it does, we destroy the PlayerAttack object so that it disappears from the screen.

SAVE + PLAY

Now, our PlayerAttack objects spawn some distance from the player and disappear after some time!

In the next chapter, we'll look at how we can make the FlyEnemy get destroyed when they are hit by the Player's attacks.

08.
CHAPTER

DESTROYING THE ENEMY

In the last chapter, we worked on the Player's sword slash attacks. Now, we will use these attacks to destroy the FlyEnemy.

As we have been doing so far, we will need to use a get_collision function to see if a collision between a Player object and a FlyEnemy happens. So, let's go to our FlyEnemy class!

FlyEnemy | Loop

```
self.x = self.x - 1

swordHit = get_collision(self, "PlayerAttack")
if swordHit:
    destroy(self)
```

Here, we create a variable called swordHit which stores collision information about the FlyEnemy and a PlayerAttack object.

Then we check to see if this variable recorded a collision with a PlayerAttack object. If so, we destroy the FlyEnemy object.

SAVE + PLAY

Now, your FlyEnemy (wasp) should disappear from the screen when it collides with a PlayerAttack (sword slash) object. Later, we will add health to our FlyEnemy so that they don't get destroyed right away.

FLYENEMY DIRECTION AND SPEED

Let's now make our FlyEnemy have a speed and a direction to move towards.

To do this, we'll create two new variables in the FlyEnemy class to control its direction and speed.

FlyEnemy | Start

```
self.sprite = sprite("wasp.png")

self.direction = "left"

self.speed = 1
```

Here, we created two variables that set the FlyEnemy's speed and direction.

SAVE + PLAY

Let's now add code in the FlyEnemy's loop tab to make the FlyEnemy move in the direction they are assigned.

FlyEnemy | Loop

```
self.x -= 1

swordHit = get_collision(self,"PlayerAttack")
if swordHit:
    destroy(self)

if self.direction == "up":
    self.y = self.y + self.speed
if self.direction == "down":
    self.y = self.y - self.speed
if self.direction == "left":
    self.x = self.x - self.speed
    self.scaleX = 1
if self.direction == "right":
    self.x = self.x + self.speed
    self.scaleX = -1
```

Here, we write some if statements in our FlyEnemy's Loop tab to check the value of the direction variable.

Then depending on the variable's value, we use the speed variable that we've created earlier to move the FlyEnemy in the direction that matches the self.direction variable. To do this, we modify either the or x value of the FlyEnemy.

Note that we've removed the line of code that reads: self.x -= 1. This is because our if statements will take care of moving the FlyEnemy in the appropriate direction.

We also change the FlyEnemy's scaleX to flip the sprite to whichever side the enemy is moving to - left or right.

SAVE + PLAY

Great! Now our FlyEnemy objects should have a speed and directions in which they fly. Only thing missing is to create more FlyEnemy Objects. We'll do this inside the Game class.

Game Start

```
...

self.wasp = FlyEnemy()
self.wasp.x = 400
self.wasp.y = -200

self.mario = NPC()
self.mario.y = -250

self.wasp2 = FlyEnemy()
self.wasp2.x = -400
self.wasp2.y = 200
self.wasp2.speed = 2
self.wasp2.direction = "right"

self.wasp3 = FlyEnemy()
self.wasp3.x = -300
self.wasp3.y = -100
self.wasp3.direction = "up"
```

Here, we've created 2 more FlyEnemy objects with their own speeds and starting directions.

SAVE + PLAY

You should now see 3 different FlyEnemy objects flying in the directions and at the speeds that we've set.

In the next chapter, we will focus on the Player's health so that they don't get destroyed right away!

09.
CHAPTER

PLAYER HEALTH

In this chapter, we will give our Player health so that they have a better chance of winning the game. We will also give them a chance to recover by using an invisibility timer that blocks FlyEnemy objects from damaging the Player for a short period of time. Finally, we'll make our FlyEnemy drop some loot when they are destroyed!

Let's start by adding a health variable to our Player.

Player | Start

```
self.sprite = sprite("pixelheadRight.png")

self.money = 0

self.hasSword = False

self.direction = "right"

self.health = 3
```

Here, we've created two new variables inside the Player's Start tab. The first is a health variable to keep track of the Player's health.

SAVE + PLAY

We will now update and use these two newly created variables to help the Player not get destroyed right away when they collide with a FlyEnemy and to give them a window of time in which they do not get hit after a collision with a FlyEnemy.

Player | Loop

```
...

oreHit = get_collision(self, "Collectible")
if oreHit:
    destroy(oreHit)
    self.money = self.money + 1
    print("Money: " + str(self.money))

waspHit = get_collision(self, "FlyEnemy")
if waspHit:
    destroy(self)
    self.health -= 1
    destroy(waspHit)
    print("Health: " + str(self.health))

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
        swordSlash.x = self.x
        swordSlash.y = self.y
        if self.direction == "up":
            swordSlash.angle = 90
            swordSlash.y += 80
        if self.direction == "down":
            swordSlash.angle = -90
            swordSlash.y -= 80
        if self.direction == "left":
            swordSlash.angle = 180
            swordSlash.x -= 80
        if self.direction == "right":
            swordSlash.x += 80
```

First, we remove the code that destroys the Player when they collide with a FlyEnemy. Then, we reduce the Player's health by 1 and destroy the wasp that has hit us.

Last, we print the amount of health we still have.

SAVE + PLAY

So far, nothing really happens when we get hit by a FlyEnemy besides destroying it. This is because we haven't really told the game to destroy the Player when their health drops below 1. Let's head over to our Player Class and do that!

Player Loop

```
...

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
        swordSlash.x = self.x
        swordSlash.y = self.y
        if self.direction == "up":
            swordSlash.angle = 90
            swordSlash.y += 80
        if self.direction == "down":
            swordSlash.angle = -90
            swordSlash.y -= 80
        if self.direction == "left":
            swordSlash.angle = 180
            swordSlash.x -= 80
        if self.direction == "right":
            swordSlash.x += 80

if self.health <= 0:
    destroy(self)
```

Here, we've added an if statement that checks to see if the Player's health variable has dropped to 0 or below. If it has, we destroy the Player object.

SAVE + PLAY

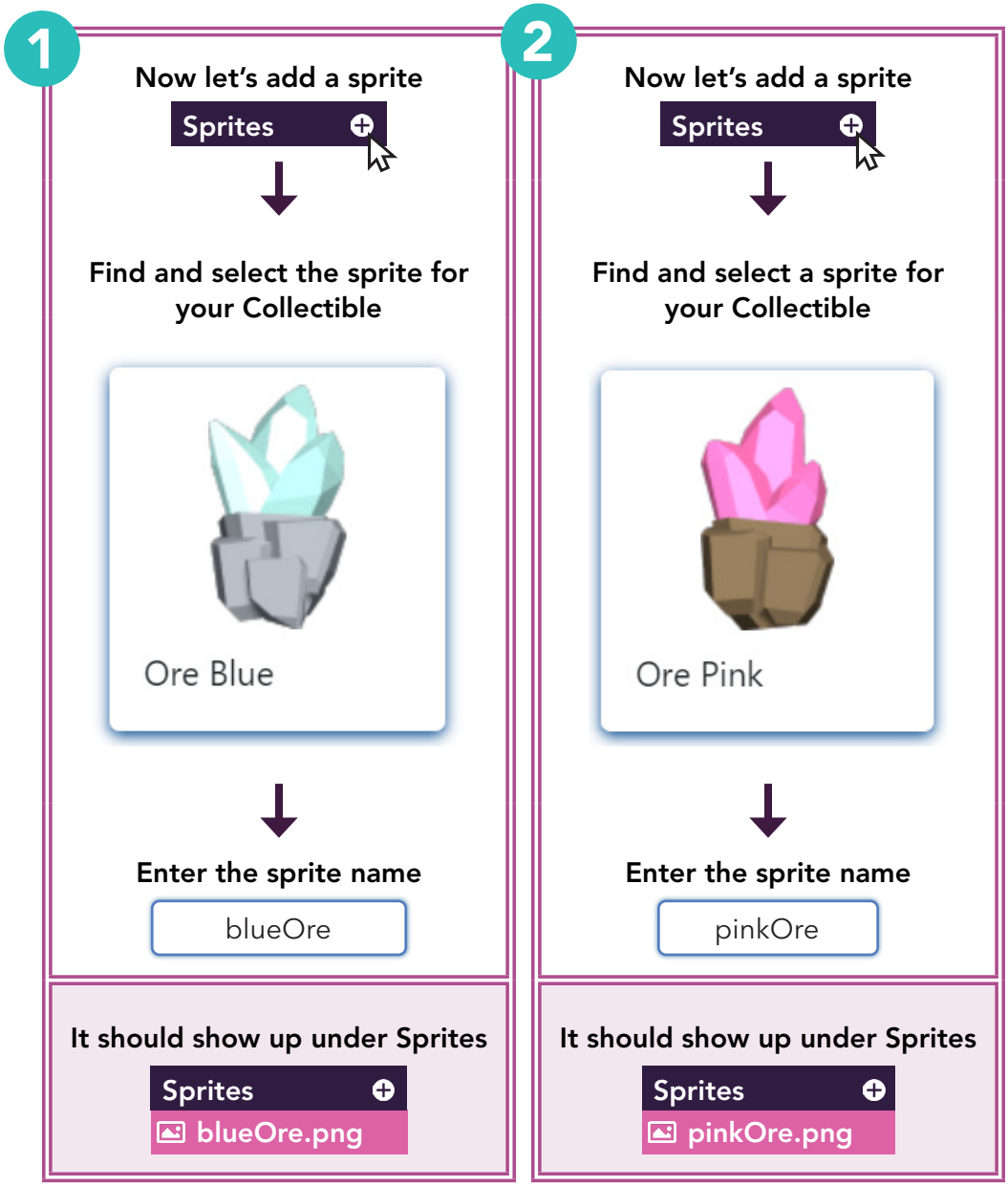
Great! Now our player's health drops when they collide with a FlyEnemy until it reaches 0, in which case, they get destroyed and vanish from the screen.

ENEMY LOOT

We will now make our FlyEnemy drop some loot for the Player once they get destroyed by a sword slash!

The FlyEnemy will drop loot at random, meaning we do not know if they will drop an item and what item it will be.

To do this, we will use our Collectible class we have. First we need to make this Collectible to be a random ore.



Collectible Start

```
import random

self.sprite = sprite("yellowore.png")

oreNum = random.randint(1,3)

if oreNum == 1:
    self.sprite = sprite("yellowore.png")
if oreNum == 2:
    self.sprite = sprite("pinkOre.png")
if oreNum == 3:
    self.sprite = sprite("blueOre.png")
```

Here, we import Python's random library by writing the code import random.

Then we use the random library to generate a random number between 1 and 3.

Next, we set the sprite of this object to one of our ores' sprites, depending on which number we got from the random function.

Next, we will use the same Python's random library in our FlyEnemy. We will use this function to decide whether or not the FlyEnemy will drop a loot.

FlyEnemy | Loop

```
import random

swordHit = get_collision(self,"PlayerAttack")
if swordHit:
    lootChance = random.randint(0,100)
    if lootChance > 50:
        loot = Collectible()
        loot.x = self.x
        loot.y = self.y
    destroy(self)

if self.direction == "up":
    self.y = self.y + self.speed
if self.direction == "down":
    self.y = self.y - self.speed
if self.direction == "left":
    self.x = self.x - self.speed
    self.scaleX = 1
if self.direction == "right":
    self.x = self.x + self.speed
    self.scaleX = -1
```

Here, we import Python's random library by writing the code import random.

Then we use the random library to generate a random number between 0 and 100.

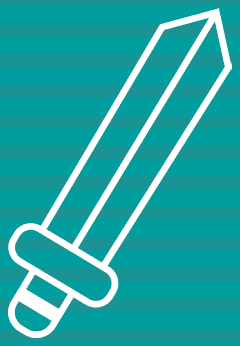
Next, inside an if statement, we check to see if the value of this random number is greater than 50. If it is, we make our FlyEnemy drop a loot in the form of a gem.

SAVE + PLAY

Great! notice that FlyEnemy doesn't always drop a loot when they're destroyed. This is because we only want them to drop a loot if the random number is greater than 50. This is almost like saying there is a 50% chance that the FlyEnemy will drop a loot.

Go ahead and try destroying all FlyEnemy objects to see how often they drop a loot!

10.
CHAPTER



ENEMY HEALTH

In this chapter, we'll add health to our FlyEnemy so that they do not die right a way!

Similar to what we did with our Player when we added health to them, we will create two new variables inside the FlyEnemy to be its health and invisibility timer.

FlyEnemy | Start

```
self.sprite = sprite("wasp.png")

self.direction = "left"

self.speed = 1

self.health = 3
self.invTimer = 0
```

Here, we created a variable called health that will store the FlyEnemy's health.

The second variable is an invincibility timer variable (invTimer) which will give our enemy some breathing space after the player hits them, so that they can't be hit right away again.

Note, again, that the variable invTimer is short for invincibility timer. We could have named it invincibilityTimer instead. However, it is a good habit to avoid long variable names when possible.

Let's now change the FlyEnemy's Loop tab!

FlyEnemy | Loop

```
import random

self.invTimer -= 1

swordHit = get_collision(self,"PlayerAttack")
if swordHit:
    if self.invTimer <= 0:
        self.health -= 1
        self.invTimer = 10
    lootChance = random.randint(0,100)
    if lootChance > 50:
        loot = Collectible()
        loot.x = self.x
        loot.y = self.y
    destroy(self)

if self.direction == "up":
    self.y = self.y + self.speed
if self.direction == "down":
    self.y = self.y - self.speed
if self.direction == "left":
    self.x = self.x - self.speed
    self.scaleX = 1
if self.direction == "right":
    self.x = self.x + self.speed
    self.scaleX = -1

if self.health <= 0:
    lootChance = random.randint(0,100)
    if lootChance > 50:
        loot = Collectible()
        loot.x = self.x
        loot.y = self.y
    destroy(self)
```

Here, we first keep counting down our invTimer.

Then, we remove all the loot code and destroy(self) so that the FlyEnemy does not disappear right away after a collision. Instead, we reduce the enemy's life by one and set the invTimer to 10 if invTimer is already less than 0.

When we set the invTimer to 10, it means that our enemy will need to wait 10 frames to get hit again. In this way, a single attack won't instantly destroy our enemy.

Next, At the end of our script we check if the enemy's health is less or equal than zero. If it is, we then calculate the loot chance and destroy the enemy.

SAVE + PLAY

Now our FlyEnemy doesn't get destroyed right away. Their health allows them to take a few hits before getting destroyed.

11.
CHAPTER

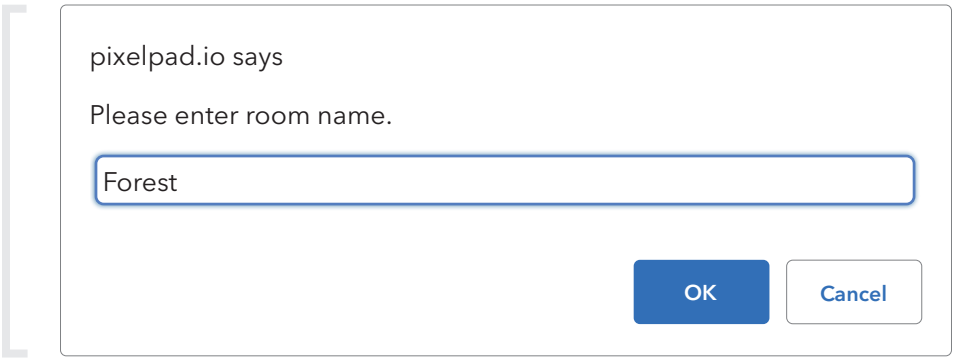
ROOMS

Rooms will help us create new levels/scenes so that our Player can travel between them.

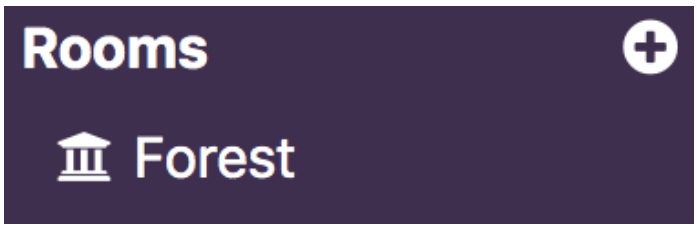
We'll first start by creating a new Room which will be our second scene/level.

To create a new room you simply click on the plus icon next to Rooms.

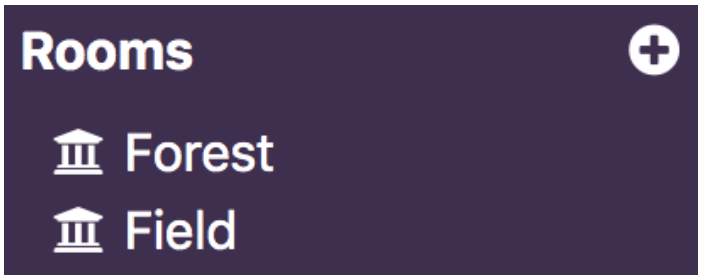
You will then be prompted to give your Room a name. Name it Forest and then click on the blue OK button.



You should now see a new room under Rooms.



Add another room to your project. Name it "Field"



Now, we will move all of the code that we have inside the Game class except the code that creates the Player to our Forest Room. To do this, we need to:

Open the Game Class and go to the Start tab.

Highlight the code with your mouse, and copy it with Ctrl+C

Leave "self.pixelhead = Player()" in Game Class.

Game Start

```
self.forest = Background()

self.pixelhead = Player()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

self.ore2 = Collectible()
self.ore2.x = -250
self.ore2.y = -80

self.wasp = FlyEnemy()
self.wasp.x = 400
self.wasp.y = -200

self.mario = NPC()
self.mario.y = -250

self.wasp2 = FlyEnemy()
self.wasp2.x = -400
self.wasp2.y = 200
self.wasp2.speed = 2
self.wasp2.direction = "right"

self.wasp3 = FlyEnemy()
self.wasp3.x = -300
self.wasp3.y = -100
self.wasp3.direction = "up"
```

Here, we copy all the code in bold and paste it in our Forest Room. Once that's done, we can remove the copied code from the Game Class.

The only code you should have in your game class right now is the code that creates the Player.

Now, we'll paste this code inside our Forest Room.

Open the Level1 Class and go to the Start tab.

Paste your code that you copied back in the Game class with Ctrl+V

Forest | Start

```
self.forest = Background()

self.ore1 = Collectible()
self.ore1.x = 300
self.ore1.y = 100

self.ore2 = Collectible()
self.ore2.x = -250
self.ore2.y = -80

self.wasp = FlyEnemy()
self.wasp.x = 400
self.wasp.y = -200

self.mario = NPC()
self.mario.y = -250

self.wasp2 = FlyEnemy()
self.wasp2.x = -400
self.wasp2.y = 200
self.wasp2.speed = 2
self.wasp2.direction = "right"

self.wasp3 = FlyEnemy()
self.wasp3.x = -300
self.wasp3.y = -100
self.wasp3.direction = "up"
```

Here, we've pasted all of the code that was in our Game class's start tab (except the code that creates the Player object) in the start tab of our Room class.

SAVE + PLAY

If you hit Play, you'll notice that our background, FlyEnemies, and ores are gone. This is because we haven't written any code to take the user to the Forest Room/level.

Let's do that now!

Game Start

```
self.pixelhead = Player()

set_room("Forest")
```

Here, we use the set_room() function to switch to the Forest room.

SAVE + PLAY

Our background, NPC, FlyEnemies and Coins are visible again but our Player isn't!

To make sure that our Player is in the Forest room we need to use what's called persistence. That is, we need to make our Player "persistent" to prevent it from being destroyed when we load a new room! We will also need to modify its z axis so that once the new room is loaded, it doesn't get overlapped by the background.

Player | Start

```
self.sprite = sprite("pixelheadRight.png")

self.money = 0

self.hasSword = False

self.direction = "right"

self.health = 3

self.persistent = True
self.z = 1
```

Here, we set our Player's persistent value to True to make sure that it doesn't get destroyed once a new room is loaded.

We also set the z axis to 1 to make sure that the pixelhead appears over the background and not behind it once the new room is loaded.

SAVE + PLAY

We can now see and interact with our Player in the Game like we used to before!

12.
CHAPTER

FIELD ROOM

In this chapter, we will make the Player travel between different rooms once they reach one end of the screen.

Inside the Forest Room's loop, we'll check the Player's x value and see if it has exceeded a certain value. If it has, we will load a new Room that the Player can travel to.

Forest | Loop

```
if game.pixelhead.x > 600:
    set_room("Field")
```

Here, we access the Player from our Game class and check to see if its x value has exceeded 600. If it has, we simply use the set_room() function to load a new room.

SAVE + PLAY

You should now be able to change to the Field room once you move to the right of the first level. Your game will turn black because we don't have anything created in our Field room yet.

Before we work on our Field room, notice that our Player is spawned at the right end of the screen as soon as they switch rooms.

Let's write some code to make the Player appear at the far left end of the screen.

Forest | Loop

```
if game.pixelhead.x > 600:
    game.pixelhead.x = -600
    set_room("Field")
```

Here, we tell the game to place our Player at an x position of -600, which is the left of the screen.

SAVE + PLAY

Great! Our Player can now smoothly switch to the second Room/level!

Before we end this chapter, we will fix one more bug that we have.

```
'Player' object has no attribute '_x' in Game.start() on line 1
'Player' object has no attribute '_x' in Game.start() on line 1
'Player' object has no attribute '_x' in Game.start() on line 1
```

If you try to let your Player get destroyed, you will see the above error in your console.

This happens because we are checking the Player's x position all the time in the Forest loop, but when we destroy the Player, we can't access that x position any more. This causes the computer to be confused.

Let's fix this problem by resetting the game instead of destroying the player. Resetting the game means to return the game to the start point. To reset our game:

- The player needs to go to the center of the screen
- The player should have 0 money and 3 health
- The player shouldn't have a sword
- The room should be the Field room

Player | Loop

```
...

if self.hasSword == True:
    if key_was_pressed("F"):
        swordSlash = PlayerAttack()
        swordSlash.x = self.x
        swordSlash.y = self.y
        if self.direction == "up":
            swordSlash.angle = 90
            swordSlash.y += 80
        if self.direction == "down":
            swordSlash.angle = -90
            swordSlash.y -= 80
        if self.direction == "left":
            swordSlash.angle = 180
            swordSlash.x -= 80
        if self.direction == "right":
            swordSlash.x += 80

if self.health <= 0:
    destroy(self)
    self.money = 0
    self.hasSword = False
    self.health = 3
    self.x = 0
    self.y = 0
    set_room("Forest")
```

Here, we've removed the code that destroys the Player and replaced it with code that takes the Player back to the starting room/level. We've reseted the Player's health to 3, money to 0, and hasSword to false. Then, we changed its x and y values back to (0,0), which is our Player's starting position once the game starts and setted the room to be the Field room.

SAVE + PLAY

Great! Now, If we let our Player get destroyed, they will go back to the centre of the screen with full health, giving the impression that they've just respawned.

Next up, we will add a background to our Field Room and we'll allow the Player to move back to the Field Room once they reach the left end of the screen.

Now let's add a sprite

Sprites ⊕

Find and select a sprite for your Field Room

Enter the sprite name

field

It should show up under Sprites

Sprites ⊕

field.png

Field Start

```
self.field = Background()
self.field.sprite = sprite("field.png")
```

Here, we've created a Background object that we named field and attached the "field.png" sprite to it.

Field Loop

```
if game.pixelhead.x < -600:
    game.pixelhead.x = 600
    set_room("Forest")
```

Here, similar to what we did before in the Field Room, we've added an if statement that checks to see if the Player's x position is less than -600. If it is, we go back to the Field Room. Finally, we set the Player's position to 600 so that they can have a smooth transition between rooms.

SAVE + PLAY

You should now be able to smoothly walk between the Field and Forest Rooms!

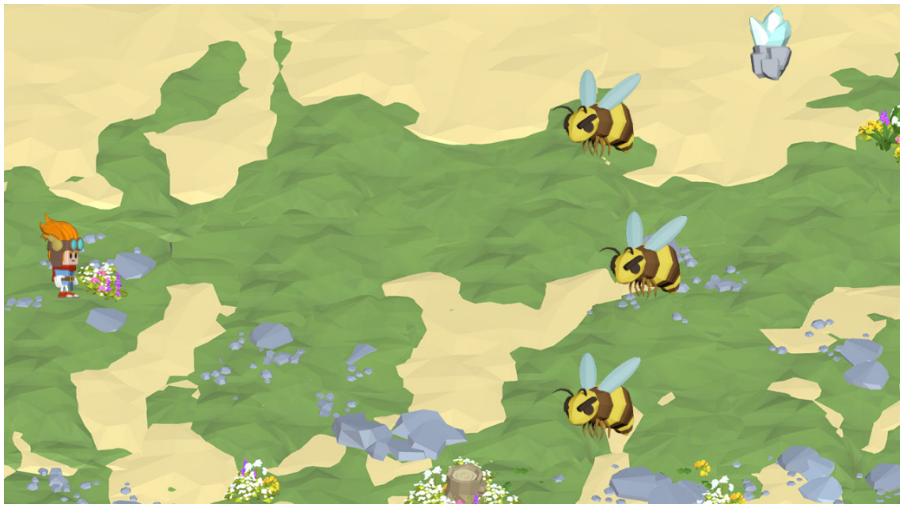
Now, let's fill our Field room with whatever objects we want! Feel free to come up with your own code!

Field | Loop

```
self.field = Background()
self.field.sprite = sprite("field.png")

self.ore2 = Collectible()
self.ore2.x = 450
self.ore2.y = 300

self.wasp = FlyEnemy()
self.wasp.x = 400
self.wasp.speed = 2

self.wasp2 = FlyEnemy()
self.wasp2.x = 400
self.wasp2.y = 200
self.wasp2.speed = 3

self.wasp3 = FlyEnemy()
self.wasp3.x = 400
self.wasp3.y = -200
self.wasp3.speed = 3
```

Here, I filled up my Field room with 3 enemies and an ore!

SAVE + PLAY



You can continue this infinitely! Make Level3, Level4, Level5 and so on and so forth. Try to make each one a little more challenging than the last one.

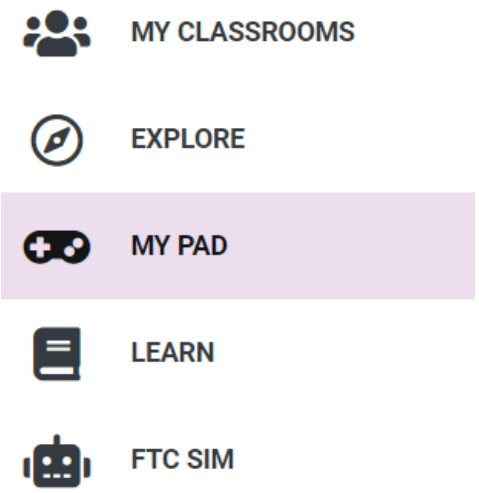
Maybe in some of the levels it's like a break room with only 1 enemy but a bunch of ores to collect. It's up to you!

Congratulations on completing this course! You can always keep working on your game. You just need to log into your PixelPAD account from any other computer connected to the internet!

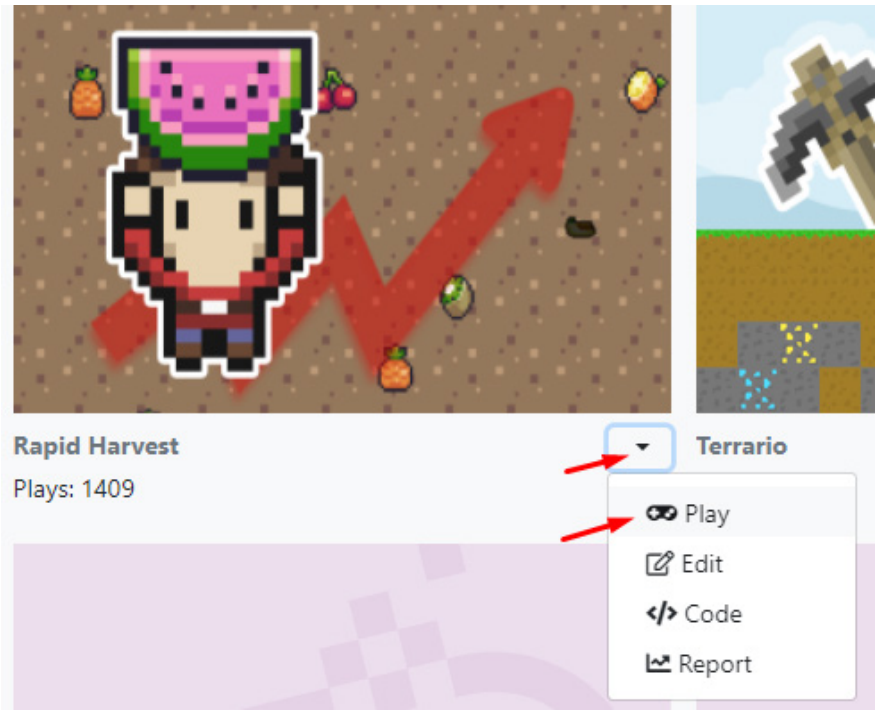
SHARING MY GAME

If you want to share your game with your friends, you just have to follow these simple steps:

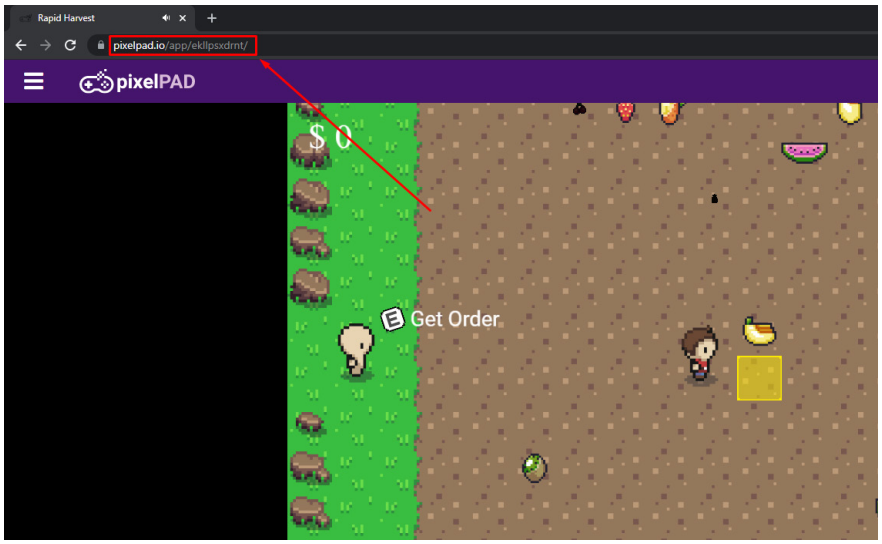
1. Go to the MyPad Section



2. Find the game you want to send to your friends and click on the triangle beside the game's name. Then, click Play.



3. Now you just need to find this page's link to send to your friends. You can easily find it at the top of your browser window. Simply copy that link and send it to your friends.



4. Done! Your friends should now be able to play your game!

EXTRA ACTIVITIES

The following activities are optional and should be added to your current game. Most of them can be added during your game's development, but some might require your game to be already completed. You can check the prerequisite chapters beside the activities to know if you are able or not to do it at the stage you are now in the course.

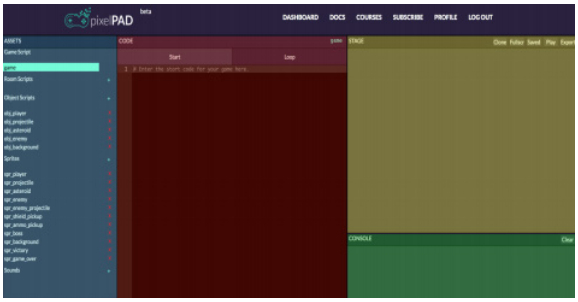
#	Prerequisite	Activity
1	Chapter 5	Add another enemy to your game. › Add a GroundEnemy (a golem)
2	Chapter 6	Create an NPC that will give you 3 money when you talk to him.
3	Chapter 8	Keep working with the golem added in the Activity #1: › Make the golem to move slower than a FlyEnemy › Make the golem to follow the player arround
4	Chapter 9	Add an NPC that will increase your health to 3 when you talk to him.
5	Chapter 12	Add a "MainMenu" room to your game › When you press Play, your game starts in that room › When you press the space key, the MainMenu room sets the Level1 room as the current room (starts the game) › Add a cool background to your MainMenu room

6	Chapter 12	Add an NPC in your game that will teleport you to a random room
7	Chapter 12	Add a boss room to your game with a new Boss enemy that behaves differently than the other enemies we already have in our game.

GLOSSARY

WHAT IS PIXELPAD?

PixelPAD is an online platform we will be using to create our own apps or games!



The PixelPAD IDE is composed of 4 areas:

ASSETS: Your assets are where you can add and access your classes and sprites. Classes are step-by-step instructions that are unique to the object. For example, the instructions for how your player moves will be different from the way your asteroid moves! Sprites is another word for image, and these images give your objects an appearance!

CODE: In this section, you will write instructions for your game. To write your code, click within the black box and on the line you want to type on. To make a new line, click the end of the previous line and then press "Enter" on your keyboard.

STAGE: The stage is where your game will show up after you write your code and click Play (or Stop and then Play). Don't forget to click save after you make changes to your code!

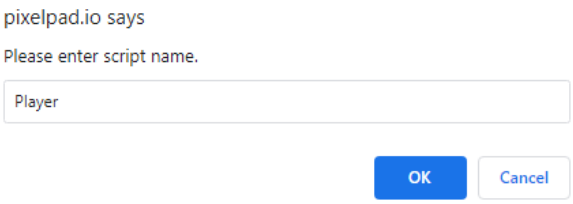
CONSOLE: Your console is where you will see messages when there are errors in your code, and also where you can have messages from your game show up such as the score, or instructions on how to play your game.

SCRIPTS

Scripts and Assets

Two of the asset types, rooms and classes, are script assets. Script assets (or scripts) are assets that have code inside them. Sprites are not considered scripts, because they do not contain any code.

Creating Scripts and Assets: To Create an asset, you start by clicking the + next to "Rooms", "Classes" or "Sprites"



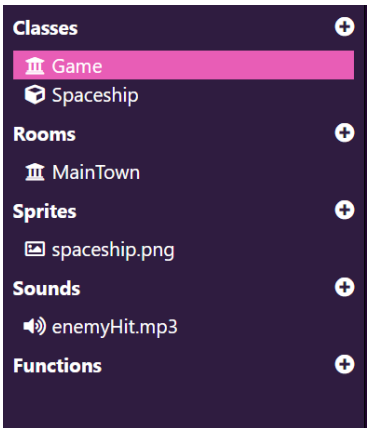
Then type in any name you'd like. My particular convention looks like this:

- "MainTown" -> room
- "Player" -> player class
- "background" -> background sprite
- "stepGrass" -> steps sound effect

Classes and rooms (scripts) should follow the "TitleCase" standard, where all words are capitalized. For sprites and sounds (assets) we use the "camelCase" standard, where the first word is lowercase, and every word that follows is capitalized. This isn't necessary, but keeps your code neat and readable.

The Game Class

There is one class that always exists in every project: the game class. The purpose of the game class is to load all of the other assets in our project. The game class represents our entire game.



DEFAULT OBJECT PROPERTIES

Sprite, scaleX and scaleY

Every class inherits default properties when created in PixelPAD. The First of these few properties you should learn about are:

.sprite, .scaleX, and *.scaleY*

.sprite is the image of the object. The value of *.sprite* is an image object which we will get to later. *.scaleY* takes a float between 0 and 1 and stretches the sprite of the object lengthwise. *.scaleX* takes a float between 0 and 1 and stretches the sprite of the object widthwise.

X, Y and Z Coordinates

The position of an object is where the object is. In programming, we usually describe an object's position using a pair of numbers: its X coordinate and its Y coordinate.

An object's X coordinate tells us where the object is horizontally (left and right), and its Y coordinate tells us where the object is vertically (up and down).

[0,0] is the middle of the screen

- .x takes the value of the x position of the object. The higher .x is, the farther to the right it is.
- .y takes the value of the y position of the object. The higher .y is, the higher up the object is.
- .z takes the value of the z position of the object. The higher .z is, the closer to you the object is.

DOT NOTATION AND SELF

Dot Notation

Dot notation is like an apostrophe s ('s). Like "Timmy's ball" or "Jimmy's shoes".

The *'s* tells you who you're talking about. In code instead of using apostrophes we use dots to talk about ownership.

So when we say *player.x* we're really saying "player's x"

Examples of Dot Notation:

- *player.x* -> refers to the player's x value
- *player.scaleX* -> refers to player's x scale (percentage)

The "Self" Property

Self refers to whichever class you are currently in.

So if you're typing code inside the Spaceship class, saying "self" refers to the Spaceship class itself.

Examples of Self

```
#Code inside "Spaceship"
self.x = 50
self.y = 30
self.scaleX = 0.5
self.scaleY = 0.5
```

The code would make Spaceship move to the right by 50px, up by 30px, and reduce its image size in half.

COLLISIONS

What Are Collisions?

Think of collision as checking whenever two objects touch. In Mario, whenever he collides with a coin, it runs the code to add a score.

In PixelPAD, it's when the "bounding boxes" of sprites touch. This includes the transparent areas of the sprite as well!

We will use collisions in our game to determine when our ship is hit by obstacles, when we've collected a power-up or health refill, and when we've managed to shoot down an asteroid.

The get_collision Function

When we want to check for a collision between two objects, we use an if statement combined with a special function called get_collision.

Here is an example of a get_collision function:

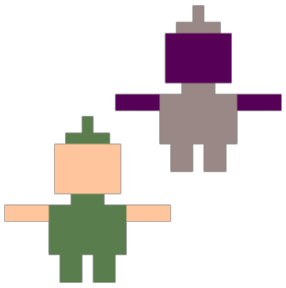
```
if get_collision(self, 'Asteroid'):
    print('The spaceship has collided with an asteroid')
```

- We start with an ordinary if statement.
- For our condition, we specify get_collision.
- We then write a pair of parentheses (()).
- Next, we write self. This specifies that we want to check for collisions against the current object.
- Finally, we write "Asteroid". This specifies the other class we want to check for collisions with. In this case, we are checking for collisions with any object created from the Asteroid class.

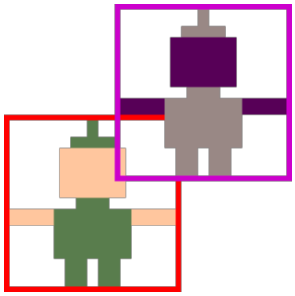
BOUNDING BOXES

Sprite's Bounds

Every object has a bounding box, which is the rectangle that contains the object's entire sprite. The get_collision condition checks for overlaps between the bounding boxes of objects, not the actual sprites. This can sometimes create surprising results. Here is an example of two objects that don't look like they should be colliding, but do:



And here they are again, with their bounding boxes shown:



DESTROYING OBJECTS

The destroy Function

When two objects collide, we generally would like to destroy at least one of them. Destroying an object removes it from the game. When an object is destroyed, it no longer exists, and trying to use it could make your game behave strangely or crash.

Destroying an object is very simple. Here is an example of destroying the player object:

```
destroy(player)
```

THE START AND THE LOOP TABS

Start and Loop

Say you decide to go for a run. You put on your runners and then you run. Running would be the loop because it repeats (one foot in front of the other), and putting runners on would be the start because it only happens in the beginning.

Similarly, in PixelPAD the "start" describes an instruction that only happens once, such as the starting position of a robot. Whereas the "loop" could describe its animation.

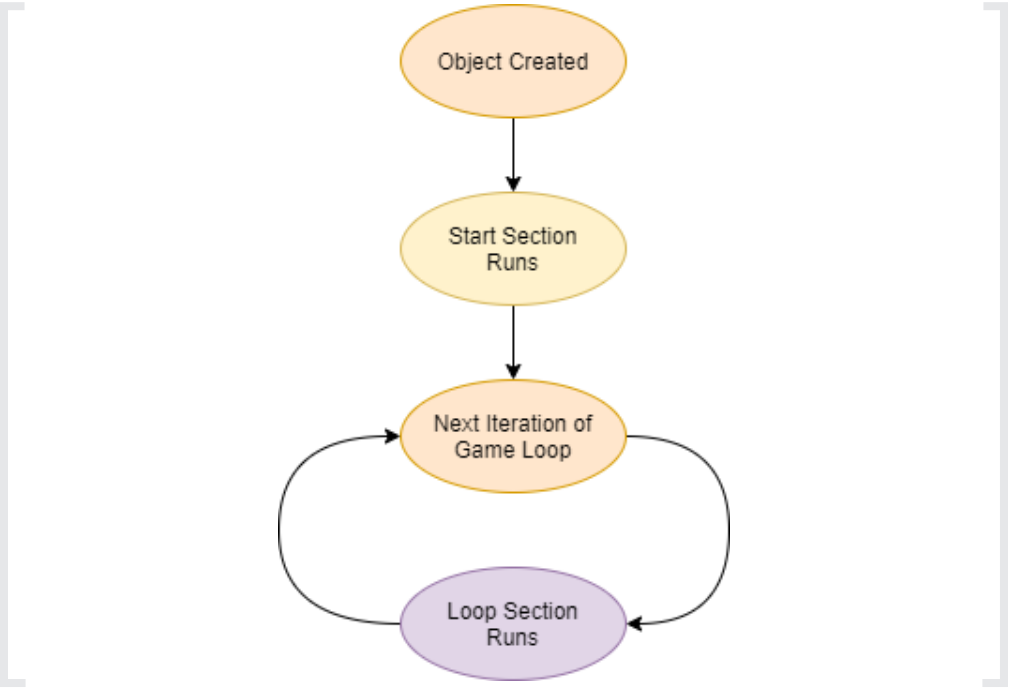
The Game Loop

Loops exist in our day-to-day life. For example, you wake up, get ready, go to school, come back home, go to sleep and repeat these things every day! So looping is the act of repeating. In programming, loops describe instructions that repeat instead of having to code each instruction again and again!.

Loops can happen every day, or they can repeat a specific number of times. For example, a programmer can code a robot to jump 100 times, or code the robot to keep jumping forever!

Video games are built around a game loop. Specifically for PixelPAD, our game loop runs the code 60 times every second!

The loop starts when we click the Play button, and stops when we click the Stop button. It goes around and around for as long as the game is playing, updating each of our objects a little bit at a time.



How Do We Use The Game Loop?

When we write code for our objects, we can choose to place it in one of two sections: the Start Section or the Loop Section. Code placed in the Start Section is executed as soon as we create the object. Code placed in the Loop Section, however, is added to the game loop, which means it will be executed over and over until the game stops.

CONDITIONALS

Conditions

So far, whenever we've written any code, all we've done is give the computer a list of commands to do one after the other. Using conditions, we can tell the computer to make a decision between doing one thing or another.

If Statements

The way we write conditions in our code is by using if statements. Here is an example of a simple if statement:

```
if key_is_pressed('D'):
    self.x = self.x + 1
```

- Start with the word if
- Next, we write our condition. The condition of an if statement is a true or false question that we ask the computer to answer for us. In the above example, our condition is key_is_pressed('D'), which is asking, "Is the D key being pressed?"
- After the condition, we write a full colon (:), and then make a new line.
- Next, we indent our code, which means we start typing it a little bit further to the right than we normally would
- Finally, we write the body of the if statement. If the condition of the if statement turns out to be true, then the computer will run whatever code we put inside the body. In the above example, the body is self.x = self.x + 1

INDENTATION IN PYTHON

Indentation

Indentation is when code is shifted to the right by adding at least two spaces to the left of the code. Indentation is important for two reasons:

- Code that is indented is considered to be part of the body of the if statement by the computer. As soon as we stop indenting the code, we are no longer inside of the if statement.
- Indentation helps us visually see the structure of our program based on the shape of our code. This helps us navigate our code and find bugs more easily.

For example:

```
if key_is_pressed('D'):
    self.x = self.x + self.speed
if key_is_pressed('A'):
    self.x = self.x - self.speed
if key_is_pressed('W'):
    self.y = self.y + self.speed
if key_is_pressed('S'):
    self.y = self.y - self.speed

if key_was_pressed(' '):
    sound_play(self.shootingSound)
    bullet = object_new('Bullet')
    bullet.x = self.x
    bullet.y = self.y
    if self.powerUp == True:
        bulletL = object_new('Bullet')
        bulletL.x = self.x - 40
        bulletL.y = self.y
        bulletR = object_new('Bullet')
        bulletR.x = self.x + 40
        bulletR.y = self.y
```

We can see clearly that the statements only run if the condition is met. E.g. the player presses the SPACE button.

It is very important that all of the code in the same body be indented using the same number of spaces on every line.

KEY PRESS

Keyboard Input

One kind of condition we can use is a keyboard check. Keyboard checks can be used to determine whether a keyboard key is being pressed or not. We can use keyboard checks to make things happen when the player presses or releases a keyboard key.

Keyboard checks are done using the key_is_pressed function. Here is an example of using the key_is_pressed function:

```
if key_is_pressed("W"):
print("You are pressing the W key!")
```

The code inside the apostrophes is the name of a keyboard key. Most keys are named the same as the letter or word on their keyboard key. A few keys have specific names:

- The space bar's name is space.
- The arrow keys are named arrowLeft, arrowRight, arrowUp, and arrowDown.
- If you have a return key, it is named enter.

COMPARISONS

Comparisons

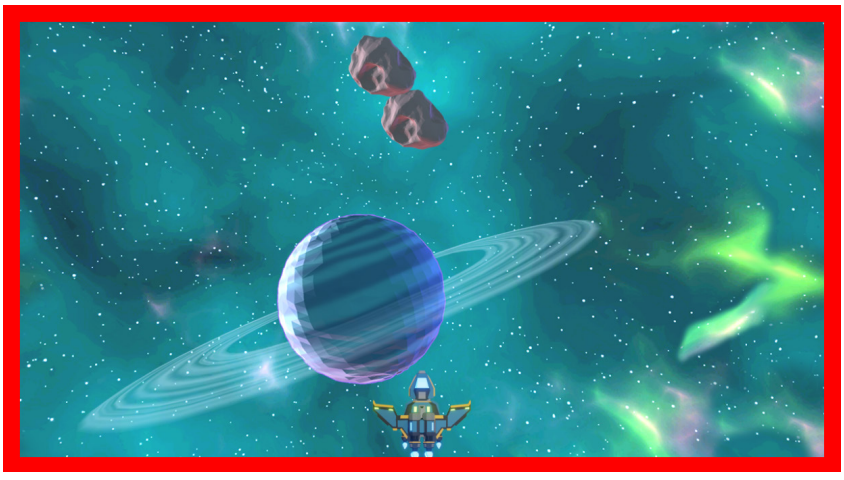
If we have code like: x > 300, this is a specific kind of condition called a comparison. Comparisons are true/false questions we can ask the computer about pairs of numbers. There are six main kinds of comparisons, each with its own operator (special symbol). This table shows an example of each kind of comparison:

Example Question	Example Code
Is x **smaller than** y?	**x < y**
Is x **bigger than** y?	**x > y**
Is x **smaller than or equal to** y?	**x <= y**
Is x **bigger than or equal to** y?	**x >= y**
Is x **equal to** y?	**x == y**
Is x **not equal to** y?	**x != y**

There are other kinds of conditions, but comparisons are the kind that we will be using most often.

Adding Boundaries Example

We can add boundaries to our game using if statements and comparisons.



In our Player class' Loop Section, add this new code at the bottom:

The Left Boundary

```
if self.x < -600:
    self.x = -600
```

The Right Boundary

```
if self.x > 600:
   self. x = 600
```

The Top Boundary

```
if self.y > 300:
    self.y = 300
```

The Bottom Boundary

```
if self.y < -300:
    self.y = -300
```

COMMENTS

Explaining Code with Comments

Computer code is complicated. That's why, a long time ago, some very smart programmers invented code comments.

Comments are like little notes that you can leave for yourself in your programs. The computer completely ignores comments in your code. You can write whatever you want inside of a comment.

How to Write a Comment

You can write a comment by starting a line of code with a pound sign, which is the # symbol (you might call this symbol a hashtag). Here is an example of some well-commented code:

```
# Shoots projectiles if the Space key is pressed
if key_was_pressed(' '):
    self.redLaser = Projectile()
    self.redLaser.x = self.x
    self.redLaser.y = self.y

# Player gets destroyed if health reaches zero
if self.health <= 0:
    destroy(self)
```

Comments are an extremely useful tool, and you should get in the habit of writing them. Comments help us remember what our code does, help others understand our code, and help us keep our code organized.

TYPES OF BAD COMMENTS

Misleading Comments

It's important to remember that comments are notes. The computer doesn't read our comments when it's deciding what to do next. Because of this, comments can sometimes be inaccurate. We should always read the code, even if it is commented, to make sure it does what we think it is doing.

Here is an example of a misleading comment:

```
# This code moves the player up when the Up Arrow key is pressed
if key_is_pressed('arrowUp'):
    self.y = self.y - 5
```

The comment says that this code makes the player move upwards, but when we read the code, it actually makes them move downwards. If we just read the comment without checking it against the code, we would have no idea why our game wasn't working properly.

Obvious Comments

Another type of bad comment is an obvious comment. Obvious comments don't add any meaningful information to your code; they usually just re-state what the code is saying in plain English. Here is an example of an obvious comment:

```
# Adds one to x
self.x = self.x + 1
```

Obvious comments clutter up our code and can slowly turn into misleading comments if we're not careful. If a comment doesn't add anything meaningful to our code, it's best to just delete it.

Vague Comments

Vague comments are comments that don't actually explain anything. Vague comments are usually written without much thought, or because the author of the comment was told to comment on their code. Here is an example of a vague comment:

```
# Grab it
if self.carryingFruit != None:
    self.carryingFruit.xToGo = self.x
    self.carryingFruit.yToGo = self.y + 30
    self.carryingFruit.z = self.z + 1
```

The comment at the top of this code just says "Grab it." It doesn't say what the code does, or how it works. Some of this code doesn't even have anything obvious to do with grabbing. Similar to obvious comments, vague comments clutter up our code and can slowly become misleading as we work on our project. It is better to just delete any vague comments you find in your code.

FRAMES PER SECOND

Frames

When you watch a movie, it looks like you're seeing one single, moving picture on the screen. This is a trick: a movie is a long series of slightly different pictures, and those pictures are being shown to you so fast that you can't tell they're individual images. Each of those single pictures is called a frame.

Games use frames, too. Every time the code in an object's Loop Section runs, the game is drawing a new frame based on where our objects are and what sprites we have attached to those objects.

Every frame in our game lasts exactly the same amount of time: 1/60th of a second. That means that there are 60 frames in a second.

Timers

A timer is a number that counts time. For example, if we were watching a clock, and counted up by one every time the clock's second hand moved, we would be timing seconds.

Since each frame in our games lasts the same amount of time, we can build a timer that counts frames by counting up by one whenever our game's Loop Section is run.

Why are timers useful? Timers let us schedule things. For example, if we wanted an asteroid to appear at the top of the screen every second, we could use a timer that counted to 60 (since each frame lasts 1/60th of a second).

RANDOM & IMPORT

Random Numbers

Most video games use some kind of randomness to change what happens in the game each time we play, to stop the game from getting boring. We can add randomness to our games using random numbers.

Random Positions

For example, whenever we create an asteroid, we've been using code like this:

```
asteroid = Asteroid()
asteroid.x = 0
asteroid.y = 300
```

This code makes asteroids appear at the top of our screen, right in the middle. When we play the game, every asteroid will appear in exactly the same place. We can change this by asking for random numbers when we set the asteroid's position:

```
asteroid = Asteroid()
asteroid.x = random.randint(-600, 600)
asteroid.y = 300
```

Random Function

random.randint is a special command which asks for a random number. The numbers between the parentheses are the smallest and largest values you want to get. For example, if we were writing a dice-rolling game, we could use random.randint(1, 6) to perform a dice roll.

Probability

Random numbers can be used to affect the probability that something will happen in your program.

For example, in the code below we're only creating an asteroid only half the time we used to by adding the random.randint(1,2) == 1 conditional.

```
if asteroid_frames >= asteroid_timer:
    asteroid_frames = 0
    if random.randint(1, 2) == 1:
        asteroid = Asteroid()
```

This code randomly chooses between the numbers 1 and 2. If it chooses 1, it creates an asteroid. If it chooses 2, it does not create an asteroid. Because the random number will be 1 half of the time, and 2 the other half of the time, this code will end up creating an asteroid half of the time as well.

Modules

When we want to use random numbers, we have to write another special command at the very beginning of our program. This is the command:

```
import random
```

This is called importing a module. Since we don't always need to use random numbers, the random.randint command is normally turned off. Importing the random module turns the random.randint command on, so we can use it.

ROOMS & PERSISTENCE

Rooms

Rooms are the big sections of our game. At the very beginning of this course, we set up a room for our game to happen in. Now that we've finished building most of our game, it's time to add a few new rooms.

Recall that when we want to change rooms, we use the room_set command. This command does two things:

1. It automatically destroys every object that was part of the previous room
2. It runs the Start Section of the new room, which should create all the objects that are part of the new room

Because each room controls all of the objects that are part of that room, each room can be used to create an independent section of our game.

Persistent Objects

Persistent objects do not belong to any room, and are never automatically destroyed by the room_set command. Persistent objects can be useful, but we have to be extremely careful to clean them up with the destroy command when we don't need them any more.

To turn an object persistent you can use the code *self.persistent = True* in the Start tab of the class

ERRORS

TYPES OF ERRORS

Compile-time Errors

Sometimes, we make mistakes when we write code. We mean to type x, but accidentally type y. We accidentally write If instead of if. These are called programmer errors.

A compile-time error is an error that results from the programmer writing code incorrectly. Another way of thinking about it is any error that produces an error message.

Compile-time errors are generally easy to find and fix, because they tend to produce detailed error messages with line numbers and file names.

Runtime Errors

On the other hand, sometimes we've written our code in the correct way, but it doesn't do what we expect it to do. For example, we could expect an object to move in one direction, but it ends up moving in the opposite direction. These are called runtime errors, and are much harder to debug.

Runtime errors occur when code is written without mistakes, but does not behave correctly.

The easiest way to find and fix runtime errors is to use the debug loop. Many problems are caused by incorrect assumptions, so make sure to always reread your code thoroughly to make sure it is doing what you think it is doing.

DEBUGGING

Debugging

Debugging is when we find and fix problems in our programs. Debugging is very important, because it's very easy to make mistakes when we write code. Even the very best programmers need to debug their code every day.

The Debug Loop

When we're fixing our programs, we can always just change code at random until our program behaves the way we want it to. If we're persistent, we can fix problems this way, but it's not a very fast (or easy!) way to work.

A better way to debug is to use the debug loop. The debug loop is a simple process that we repeat until our program works properly. This is what it looks like:

1. First, we Run our code. Running our code will let us observe it, which will show us whether there are any errors or other problems. If everything is working properly, we can stop debugging.

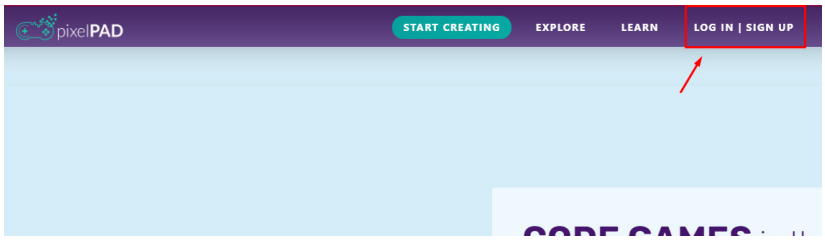
2. Next, we Read our code. Using what we learned from running our code, we look for specific commands that might be causing problems. Sometimes, an error message will tell us exactly where to look by giving us a line number and file name (for example, error in Player.Loop() on line 4 means that the 4th line of code in the Player class' loop tab is wrong). When we don't have an error message, we have to look for the problem ourselves.

3. Lastly, we Change our code a tiny little bit. Once we think we've found the source of a bug, we can change our code to either make it give us more information (this is called tracing), or we can try to fix the problem. It is important to change only a small amount of code in this step, because whenever we change our code, we risk adding new bugs to our program.

LOGGING IN

Logging onto PixelPAD

We will access PixelPAD using an internet browser such as Google Chrome, Firefox, or Safari. This way you can play and create your game from any computer! Go onto https://www.pixelpad.io



1. Click Login / Sign Up

2. Your username and password will be provided for you! If you don't have a username, please speak to one of your facilitators!

3. Click on Learn, and select the Game Tutorial you'd like to work on. This will create a blank project of a game with the tutorial open to get you started.

CHALLENGE QUESTIONS

CHAPTER 1

What is the difference between the Tutorial and My Apps section on PixelPAD?

_____

_____

_____

CHAPTER 2

What is the function that we use to create a new object?

player = _____()

For each of the following pieces of code, in what direction is the object moving in?

```
self.y = self.y + 3
```

a) Up

b) Down

c) Right

```
self.x = self.x - 7
```

a) Up

b) Right

c) Left

```
self.x = self.x - 3
self.y = self.y + 4
```

a) Up and Right

b) Up and Left

c) Down and Left

CHAPTER 3

What is the difference between key_is_pressed() and key_was_pressed?

CHAPTER 4

What other item pickups can you add to your game?

CHAPTER 5

What conditions do these nested if statements check, and in what order?

```
if get_collision(self, "Enemy"):
    if direction == "up":
        destroy(self)
```

a) Destroy self, then check if direction is up,
then check if there's a collision

b) Check if direction is up, then destroy self,
then check if there's a collision

c) Check if there's a collision,
then check if our direction is up, then destroy self

CHAPTER 6

What is the library that we import randint from?

```
import _______________
```

What conditional should you use if you only require BOTH conditions to be true in an if statement?

a) and

b) or

c) !

CHAPTER 7

If our game is running at 60 Frames per Second, how many frames is:

a) One Minute?

_____

b) One Hour?

_____

Bonus: One Year?

_____

CHAPTER 8

What is the function to change rooms?

a) room_change()

b) room_move()

c) room_set()

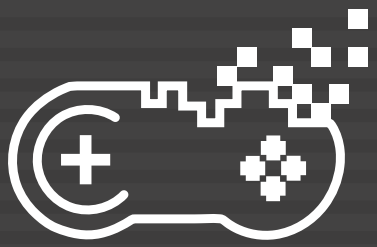
What is the purpose of creating objects and storing variables in the game script instead of room scripts?

CHAPTER 9

What other tools or items could drop from a treasure chest in your game?

How would you code it? (Explain in Words)

ERRORS GUIDE



TYPES OF ERRORS

Compile-time Errors

Sometimes, we make mistakes when we write code. We mean to type x, but accidentally type y. We accidentally write If instead of if. These are called programmer errors.

A compile-time error is an error that results from the programmer writing code incorrectly. Another way of thinking about it is any error that produces an error message.

Compile-time errors are generally easy to find and fix, because they tend to produce detailed error messages with line numbers and file names.

Runtime Errors

On the other hand, sometimes we've written our code in the correct way, but it doesn't do what we expect it to do. For example, we could expect an object to move in one direction, but it ends up moving in the opposite direction. These are calletd runtime errors, and are much harder to debug.

Runtime errors occur when code is written without mistakes, but does not behave correctly.

The easiest way to find and fix runtime errors is to use the debug loop. Many problems are caused by incorrect assumptions, so make sure to always reread your code thoroughly to make sure it is doing what you think it is doing.

DEBUGGING

Debugging

Debugging is when we find and fix problems in our programs. Debugging is very important, because it's very easy to make mistakes when we write code. Even the very best programmers need to debug their code everyday.

The Debug Loop

When we're fixing our programs, we can always just change code at random until our program behaves the way we want it to. If we're persistent, we can fix problems this way, but it's not a very fast (or easy!) way to work.

A better way to debug is to use the debug loop. The debug loop is a simple process that we repeat until our program works properly. This is what it looks like:

1. First, we Run our code. Running our code will let us observe it, which will show us whether there are any errors or other problems. If everything is working properly, we can stop debugging.

2. Next, we Read our code. Using what we learned from running our code, we look for specific commands that might be causing problems. Sometimes, an error message will tell us exactly where to look by giving us a line number and file name (for example, error in Player.Loop() on line 4 means that the 4th line of code in the Player class' loop tab is wrong). When we don't have an error message, we have to look for the problem ourselves.

3. Lastly, we Change our code a tiny little bit. Once we think we've found the source of a bug, we can change our code to either make it give us more information (this is called tracing), or we can try to fix the problem. It is important to change only a small amount of code in this step, because whenever we change our code, we risk adding new bugs to our program.

HOW TO READ ERRORS

Every PixelPAD error is separated into two parts: The error, and the error's location.

```
name 'Playr' is not defined in Game.start() on line 3
```

In this case, "Game.start() on line 3" means the error is located in the Game class, in the Start tab, on line 3.

Sometimes the console window won't point to the exact place where the error occurred. That means your error might not actually be in "Game.start() on line 3".If you think the error's location doesn't make much sense, check the last lines of code you've added to your project. That's where the error is most likely to be.

COMMON ERRORS

Name not defined

```
name 'Playr' is not defined in Game.start() on line 3
```

You're trying to access something that doesn't exist. Are you trying to instantiate (create) a class that doesn't exist? Are you trying to access a variable that doesn't exist?

Bad input

```
bad input in Game.start() on line 4
```

Your code isn't done properly. There is probably something missing or extra in there.

TypeError: Properties of Undefined

```
TypeError: Cannot read properties of undefined
(reading 'texture') in Player.Start() on Line 7
```

You're trying to load an asset (sprite, sound) that doesn't exist in your project.

<Invalid Type> object not callable

```
'<invalid type>' object is not callable in Game.start() on line 1
```

You're probably trying to set a room that doesn't exist.

Unindent does not match any outer indentation level

```
unindent does not match any outer indentation level in
Game.start() on line 2
```

There might be extra indentation (space) in front of line 2.

Object has no attribute ‘X’

```
'Spaceship' object has no attritube 'X' in Spaceship.
                    loop() on line 4
```

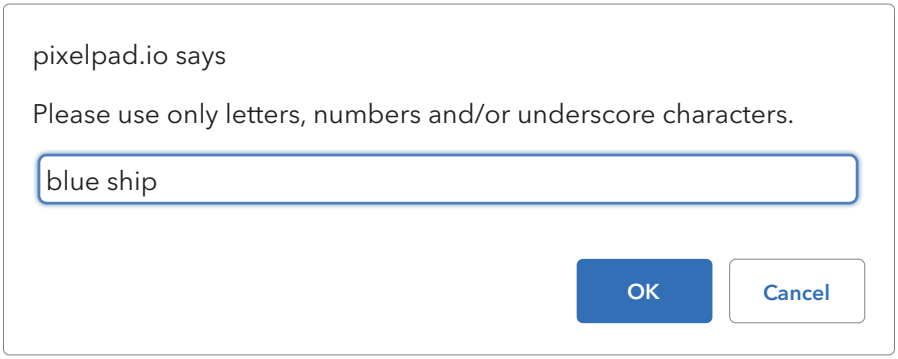
You’re trying to access the variable X inside the Spaceship object. However, that variable doesn’t exist.

Local Variable referenced before assignment

```
local variable 'sprite' reference before assignment in
                Hazard.start() on line 3
```

You’re trying to access the variable X inside the Spaceship object. However, that variable doesn’t exist.

Please use only letters, numbers and/or underscore characters.



You cannot use space or special characters when naming your classes and textures.