PYTHON

MIDDLE SCHOOL EDITION

SPACE SHOOTER

GAME DEVELOPMENT GUIDE

BY PIXELPAD

GAME DEVELOPMENT GUIDE

PIXELPAD

SPACESHOOTER 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 & COMPETENCIES

Students create a Space Shooter game, in which the user controls a spaceship and shoots asteroids and other spaceships with lasers! This is a course meant for absolute beginners and serves as an introduction to the basics of programming concepts.

Students will be able to modify the gameplay, change their sprites and personalize their game to make it uniquely their own.

COURSE GOALS

- Students have a basic understanding of computer programming and begin to understand the PixelPAD interface.
- Students finish with their own Space shooter project, customized with their own art and features.

LEARNING OUTCOMES

Computational Thinking and Algorithms

- Students understand and can describe the concept of variables
- Students can point to examples of variable usage & explain what it means
- Students can use basic PixelPAD and Python commands, including object_new(), sprite_new(), key_is_pressed()
- Students have an understanding of conditions in games, and can use this logic to modify gameplay
- Students can use the Cartesian coordinate system in PixelPAD to position their objects appropriately in the game
- Students can take simple existing code and modify it to their liking.

Creativity

Students generate ideas and explore ways to implement ideas using pre-made functions and course assets on the PixelPAD platform.

Prototyping, Testing and Debugging

Students can trial and error to make changes, solve problems, or incorporate new ideas from self or others.

Construction

Students can make a product using known procedures or through modelling of others.

Communication & Collaboration

- Students can demonstrate their product, tell the story of designing and making their product
- Students can use personal preferences to evaluate the success of their design solutions
- Students can reflect other students work and have other students reflect work

SAMPLE PROJECT

The book's project can be found here:

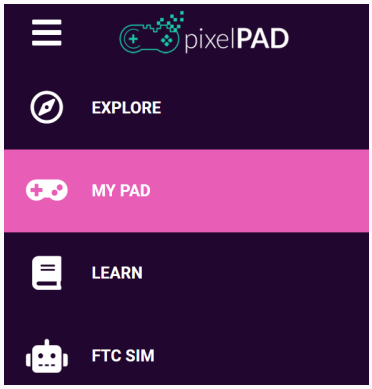
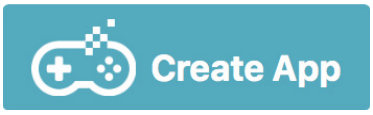
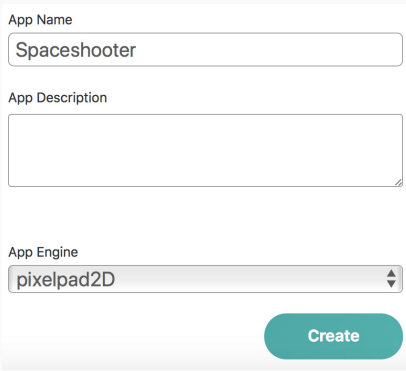
https://pixelpad.io/app/xabiacmcvyx/?edit=1

01.
CHAPTER

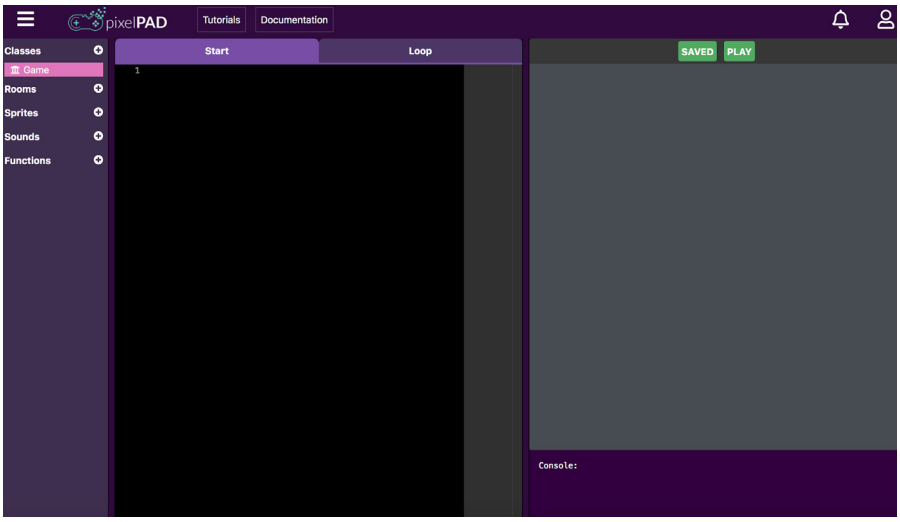
SETTING UP

In this workbook we will be making a space shooter game where our player ship will have to avoid asteroids, destroy enemy spaceships, and try to survive!

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. Here, we named it Spaceshooter. **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 BACKGROUND

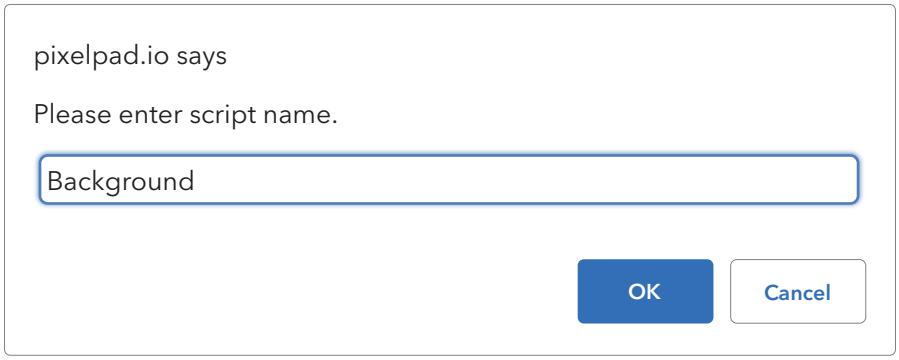
Our game right now is totally empty. Let's add the simplest first: Our background!

A background will make our game look nicer. It is simply an image that is placed at the back of our game.

The first step is to create our Background class by clicking on the white + icon next to the classes menu.

Then, a window will pop up asking us to give the class a name. Name the class Background, as shown below and make sure to capitalize the "B" in Background. 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 Background class you just created right below the Game class. The pink highlight means you have opened that class. In this image, the Background class is open.

Now, we will add some code that will make the background image 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 here! These tell you where to type in your code.

Add the following **bolded code** below.

Game | Start

```
self.space = Background()
```

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

A variable is used to store some value, this one we named space. With it, we are storing an object of type 'Background', using the command Background().

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

SAVE + PLAY

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

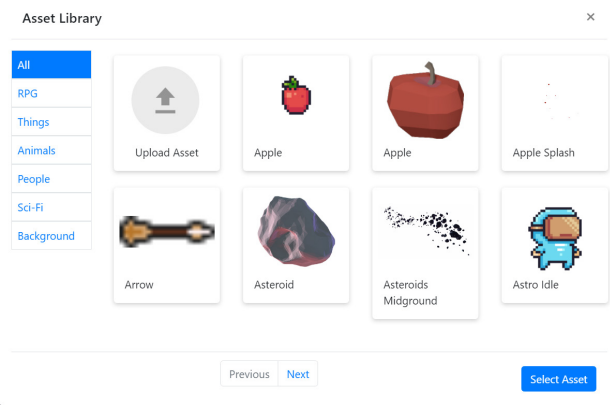
Don't be confused by them! A class is a blueprint that we use to construct/create the objects that we interact with in our game. For example, our Background class is the blueprint from which we create "actual" background objects such as our space.

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

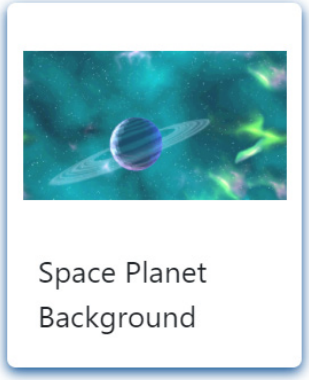
A sprite is a computer graphic that is moved on a screen or manipulated. Generally, sprites refer to 2D images that make up our game's art.

So for our Background object, we need to assign a sprite. To do this, first click on the + 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! You can click on the tabs on the left to select a category and click on Previous/Next to navigate the pages.

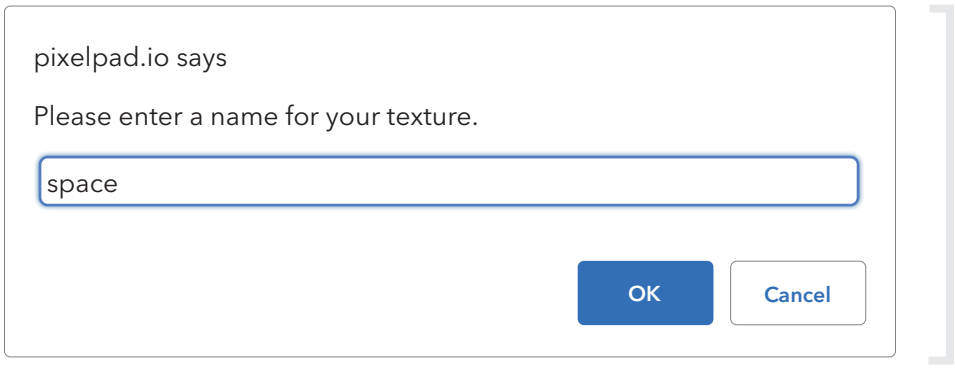


For our example, we will be using the Space Planet Background sprite. To select it, click on the background image.



Note: If you've decided to upload a sprite from your computer, make sure it has a .png/.jpg extension at the end of the name. Do not choose a moving sprite. Animations will not be covered in this project.

Once you've selected/uploaded a sprite that you like, click on Select Asset, then name your sprite "space" as shown below.



Notice here how we did not capitalize the "s" in space. This is because we are creating a sprite, not a class!

Now, you should see a sprite called "space.png" below the Sprites menu.

Ok! So, now our game has a space sprite, but we still haven't assigned it to our Background class. To do this, we will have to write some code inside of our Background class. So, let's head over there!

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

Background | Start

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

Here,we are assigning the sprite variable of our Background to space.png.

Notice that we used the word **self** one more time. "self" refers to the object running this code. We then use it to call the object's sprite. That is, the self keyword allows us to refer to the sprite that belongs to the Background object.

SAVE + PLAY

Save and Play your game to see the changes

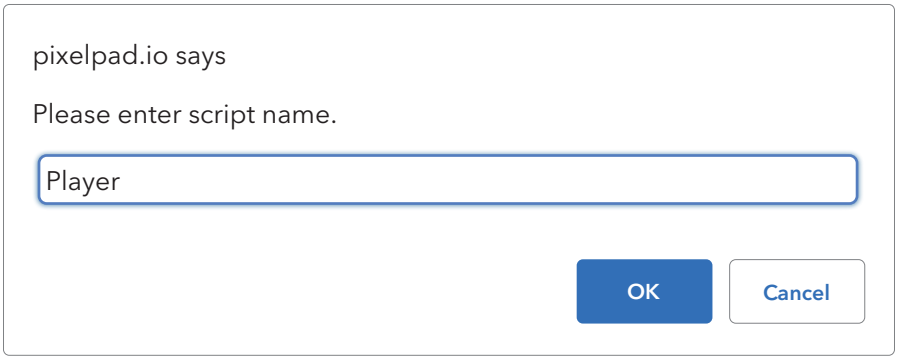
You should now be able to see your background rendered as the sprite you specified.

THE PLAYER

Now that we have our space background set up, we can add the most important piece of our game: Our player!

The first step is to create our player class just like how we did with our Background.

Then once the window pops up, we'll name the new class Player.



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.

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.space = Background()

self.hero = Player()
```

This line of code creates a copy of our Player class in the game and stores it inside the hero variable. We call the "hero" our Player object.

SAVE + PLAY

Save and Play your game to see the changes

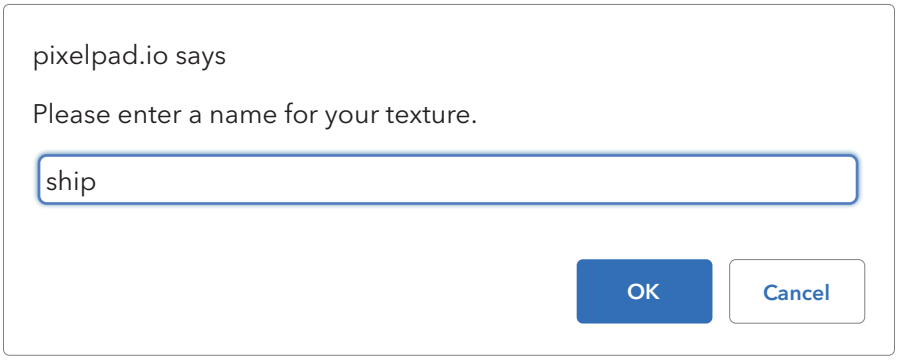
Once again you may notice that your object shows up as a blue box that says empty image… This is because we haven't assigned our object an image.

Let's add a new sprite like we did for our background by pressing the + icon besides Sprites.

For our example, we will be using the Blue Sci-Fi Player sprite.

Name your sprite “ship” as shown below and click OK.

Note: Do not use an animated sprite. We will not cover animation in this book. If you are uploading your own image, make sure the file type is **.png** and not .jpg to make sure the image background is transparent.



Ok! So now that we’ve added a ship sprite to our game we have to assign it to our player. To do this, we will have to write some code inside of our Player class. So, let’s head over there!

Player | Start

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

Here,we are assigning the new sprite we made "ship.png" to the sprite variable of Player.

SAVE + PLAY

Save and Play your game to see the changes

You should now be able to see your player rendered as the sprite you specified.

Now that we have a sprite for our player object, we can add some behaviours and functions to our ship.

MOVING THE PLAYER

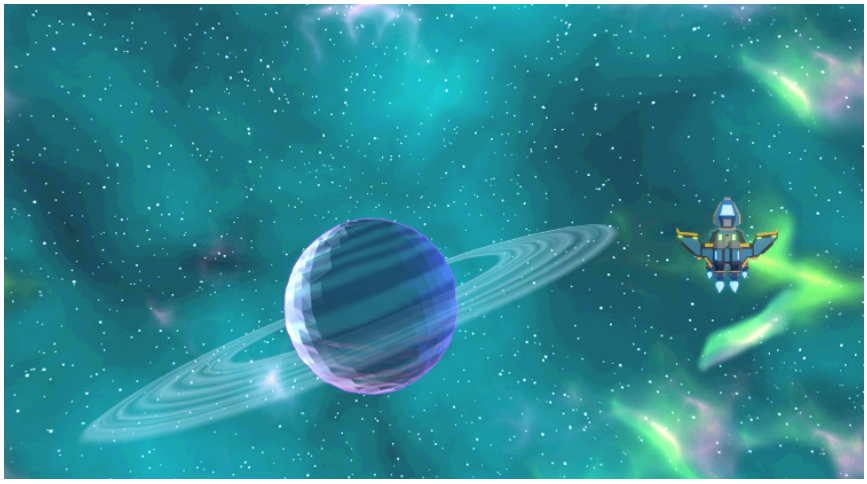
We're going to manipulate our player's position directly. Simply put, we're going to move our player! For more information on how x and y coordinates work, check out the glossary section on the Cartesian Coordinate System!

Game | Start

```
self.hero = Player()

self.hero.x = 400
```

Here,we are directly changing the x position of the hero to 400 units to the right, which is in the positive direction.

SAVE + PLAY



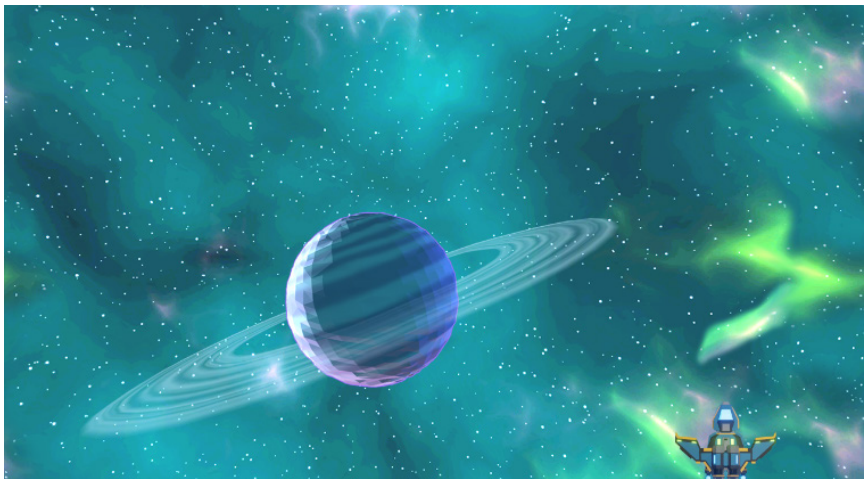
The x-axis moves our player left and right while the y-axis moves it up and down. You can also try changing the y position of your player by using self.hero.y

Game Start

```
self.hero = Player()

self.hero.x = 400
self.hero.y = -300
```

Here,we are directly changing the y position of the hero to 300 units to the bottom, which is in the negative direction.

SAVE + PLAY



You should now see that your ship moved to different positions depending on the x and y position you gave the object.

Let's change the x and y positions to place our hero at the middle of the screen, but closer to the bottom.

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

Game Start

```
self.hero = Player()

self.hero.x = 400
self.hero.y = -300

self.hero.x = 0
self.hero.y = -200
```

Here,we are directly changing the x and y positions of the hero to place it at the bottom-middle of the screen.

SAVE + PLAY

PLAYER CONTROLS

As of right now, our Player just stays in one static position. In order to avoid asteroids and enemies, we need to be able to move.

Why don't we go ahead and add controls to our game? Let's make the keys "WASD" be our movement keys! We'll be using something called conditionals to do this. For in-depth information about if statements and conditionals, head to the section on Conditionals and If Statements in the Glossary!

Notice how "self.x = self.x + 2" is **indented**? This is intentional. If you do not have an indent, you can press the Tab key on your keyboard to add it.

Player | Loop

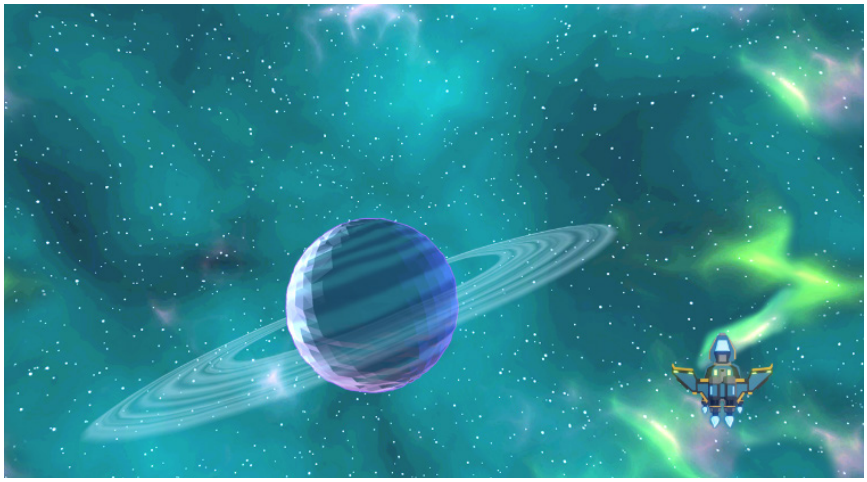
```
if key_is_pressed("D"):
  self.x = self.x + 2
```

Here,we make a conditional statement for one of the directions our Player can move in.

If the "D" key is pressed down, the player will move 2 units to the right. Since this code is placed in the loop tab, the player will keep moving while the key is pressed down.

In order to specify what happens in case the condition is met, we use indentation. **Indentation** is the space the second line has on the left.

SAVE + PLAY



Your spaceship should be able to move right when the D key is pressed now! However, we still need to make it move left with the A key as well.

Player | Loop

```
if key_is_pressed("D"):
  self.x = self.x + 2

if key_is_pressed("A"):
  self.x = self.x - 2
```

Here,we make the player move 2 units to the left if the "A" key is pressed.

SAVE + PLAY

Your spaceship should be able to move left and right by pressing the A and D keys. Now we just need to make our spaceship move up and down! To do that we are going to need to change the y position this time.

Player | Loop

```
if key_is_pressed("D"):
  self.x = self.x + 2

if key_is_pressed("A"):
  self.x = self.x - 2

if key_is_pressed("W"):
  self.y = self.y + 2

if key_is_pressed("S"):
  self.y = self.y - 2
```

Here,we add two more conditions to check whether the W or S keys are being pressed. If the W key is pressed the player moves up, and if the S key is pressed the player moves down.

SAVE + PLAY

You should now be able to control your ship in all four directions!

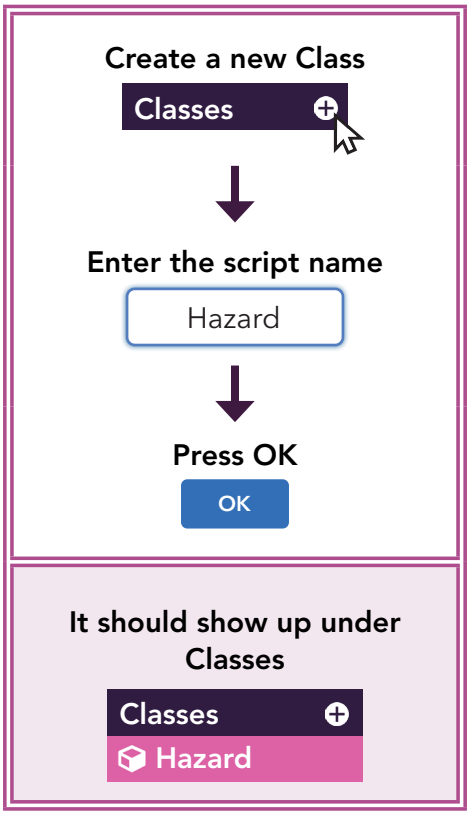
To adjust the speed that you fly at, you can change the number at the end of every if statement.

03.
CHAPTER

ASTEROIDS

Next, let's try adding some asteroids to practice what we've learned so far.

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.



After creating the Hazard class, we are ready to add it to our game. First, let's create an object named "asteroid" in our Game class.

Game Start

```
self.space = Background()

self.hero = Player()

self.hero.x = 0
self.hero.y = -200

self.asteroid = Hazard()
```

Here,we create a new object called "asteroid" that uses the Hazard class.

SAVE + PLAY

When you press play you'll notice that again, our little blue box is back. This is because there's no sprite for the asteroid. Let's add a sprite from the Asset Library!

1

Now let's add a sprite

Sprites ⊕

Find and select a sprite for your Hazard

2

Enter the sprite name

asteroid

Press OK

OK

It should show up under Sprites

Sprites ⊕

asteroid.png

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

Note: If you choosing another sprite, make sure the file type for the asteroid is .png. If you don't a white box will show up behind it.

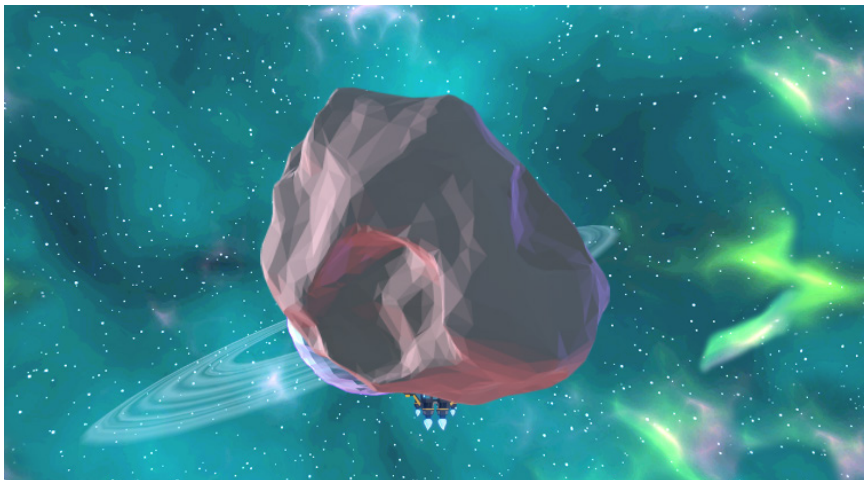
Hazard | Start

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

Here, we assign the new sprite "**asteroid.png**" to our object "asteroid" that we created back in Game Class.

SAVE + PLAY

Press play and you should now be able to see your asteroid in your scene!



SCALING

As you can see, our asteroid is too big for our game. We can make this object smaller or bigger by changing the **scale** of this object.

Every object has default **scaleX** and **scaleY** values set to 1. So, if we change the scaleX value, we can change the width of this object. If we change the scaleY, we can change the height of this object. To scale it equally, we'll have to change them both at the same time. Let's try it out!

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

Hazard | Start

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

self.scaleX = 0.2
self.scaleY = 0.2
```

Here,we're changing our scaleX and scaleY values to 0.2. We change them both at the same time so that it will scale equally. If we only change scaleX, the asteroid will look squished.

SAVE + PLAY

Notice how our asteroid got smaller. Reducing the scaleX and scaleY values makes our object smaller. Now it looks better!

MOVING ASTEROIDS

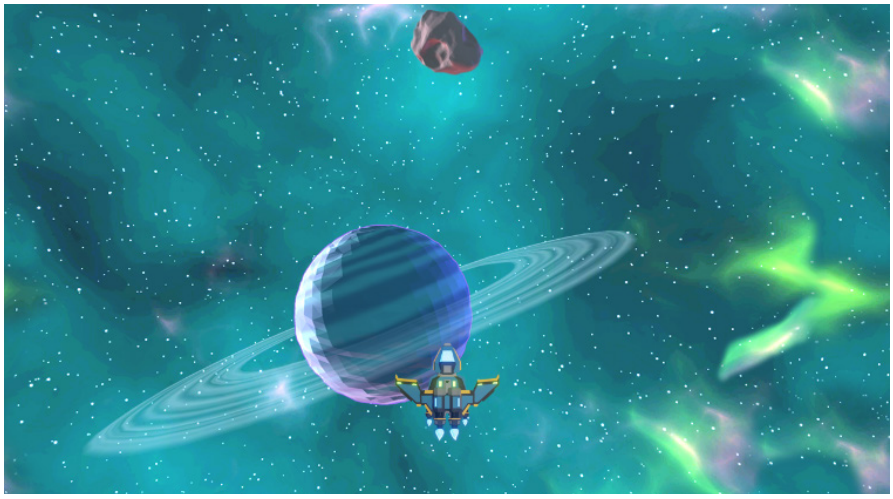
Now that our player moves, let's move our asteroid! First, let's position it at the top of the screen.

Game Start

```
self.space = Background()

self.hero = Player()

self.hero.x = 0
self.hero.y = -200

self.asteroid = Hazard()
self.asteroid.y = 300
```

Here,we move our asteroid to the top of the screen by changing its y position to 300.

SAVE + PLAY



Now, for the movement, we're going to want the asteroid to continuously move downwards. To do that, we're going to use the Loop class.

Hazard | Loop

```
self.y = self.y - 1
```

This line of code makes our asteroid go down in the y direction 1 unit every frame.

Note: We can actually simplify the above line of code to self.y -= 1. This is a simple shortcut that can be used in place of self.y = self.y - 1.

For a refresher on frames and the Loop Tab, head to "The Game Loop" section of the glossary!

SAVE + PLAY

You should see that your asteroid now moves downward continuously.

DESTROYING ASTEROIDS

As of now, our asteroids keep moving downwards forever. They disappear off the screen but they are actually still in the game because we didn't tell the game to remove them. So, we are going to want to destroy them or they will keep spawning infinitely and make the game slow.

Hazard | Loop

```
self.y = self.y - 1

if self.y < -360:
     destroy(self)
```

Here,we're going to make an if statement that checks the y position of our asteroid.

If it goes beyond -360 in the y direction, which is the bottom of our screens, we're going to destroy the asteroid because we've successfully avoided it!

SAVE + PLAY

After your asteroid has traveled to the bottom of the screen, the asteroid should disappear and be destroyed!

COLLISION

A collision is when two objects touch or overlap. We will use collisions in our game to detect when our ship is hit by obstacles, when we've collected power-ups, and when we shoot down an asteroid. Then, we'll be able to use that information to make the game do something when it successfully detects a collision!

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

Player | Loop

```
if key_is_pressed("D"):
  self.x = self.x + 2

if key_is_pressed("A"):
  self.x = self.x - 2

if key_is_pressed("W"):
  self.y = self.y + 2

if key_is_pressed("S"):
  self.y = self.y - 2

asteroidHit = get_collision(self, "Hazard")
if asteroidHit:
    destroy(asteroidHit)
    destroy(self)
```

Here, we use the variable asteroidHit to store the return value of the function get_collision.

Notice the way we've named our asteroidHit variable. We call this camel case naming. If our variable name is 2 or more words, we will "glue" these words together making sure that we capitalize the start of every new word. For example, since asteroid hit is not a valid variable name, we glue asteroid and hit and capitalize the "h" in hit i.e. asteroidHit.

The get_collision function will return the first asteroid object that collides with the player. If there is no collision, then the function will return false. This variable allows us to reference the object we collide with and destroy it inside the if statement.

Then inside our if statement, we want to destroy our player and asteroid. Later on when we introduce health we'll change what happens here, but for now, our player will be destroyed when we come into contact with an asteroid.

SAVE + PLAY

Save and Play your game to see the changes

Now when you hit an asteroid, both your spaceship and the asteroid you collided with will be destroyed!

04.
CHAPTER

TIMERS

As of right now we only have one asteroid in our game. We want a bigger challenge than just avoiding one asteroid. We want them to spawn infinitely. This will essentially make our game challenging and endless!

The first thing we need to add is a timer so that we can keep track of how much time has passed.

Game Start

```
self.space = Background()

self.hero = Player()
self.hero.x = 0
self.hero.y = -200

self.asteroid = Hazard()
self.asteroid.y = 300

self.asteroidTimer = 0
```

Here,we're creating a variable called "asteroidTimer" that will store the time that passes.

Game Loop

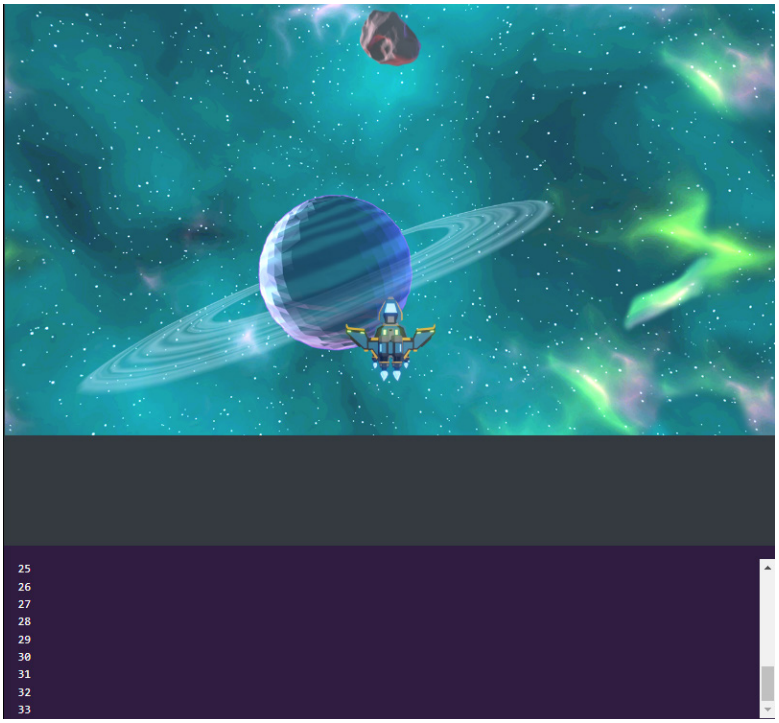
```
self.asteroidTimer = self.asteroidTimer + 1
print(self.asteroidTimer)
```

This code simply adds 1 to our asteroid timer on every frame and then prints the value of "asteroidTimer".

Notice the print() function. This function allows us to print messages to the console by wrapping the text that we wish to print inside both the single quotations and brackets. For example, print("asteroid hit!") prints and displays the text "asteroid hit" onto the console.

SAVE + PLAY

You should be able to see the asteroid timer count up in the console. The console is at the bottom of your game screen.



Notice that when the counter hits 60, one second has elapsed, you can try counting one second as you press play as well.

Now that we have a functioning timer, we can create events that trigger after a certain amount of time has elapsed.

To do this we write if statements that check if the asteroid timer is equal to or greater than a certain time interval.

Let's add a simple one now for our asteroid timer.

Game Loop

```
self.asteroidTimer = self.asteroidTimer + 1
print(self.asteroidTimer)

if self.asteroidTimer >= 120:
    self.asteroidTimer = 0
```

Here,we are saying if our self.asteroidTimer is >= 120, then reset the asteroid timer to 0. (120 is 2 seconds)

Resetting the timer to 0 will mean that we are able to endlessly recreate this timer condition. This endless behaviour will allow us to continuously spawn asteroids later.

SAVE + PLAY

SPAWNERS

Right now even though the timer resets, we don't actually spawn any asteroids.

Let's add a way to create a new asteroid every 2 seconds now in the same script.

Game Loop

```
self.asteroidTimer = self.asteroidTimer + 1
print(self.asteroidTimer)

if self.asteroidTimer >= 120:
    self.asteroidTimer = 0
    self.newAsteroid = Hazard()
    self.newAsteroid.x = 0
    self.newAsteroid.y = 400
```

Here,we are assigning a new variable called "newAsteroid" using the Hazard Class we made earlier. Then, we set it's location to the top of our screen in the middle.

SAVE + PLAY

You will notice that a continuous line of asteroids spawns every two seconds in the middle.

You can now remove the print statement from the game's loop script so you don't clutter your console.

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

Game Loop

```
self.asteroidTimer = self.asteroidTimer + 1
print(self.asteroidTimer)

...
```

Removing this stops our game from printing the time in console.

SAVE + PLAY

RANDOMNESS

You will now notice that your game continuously spawns asteroids but only in the middle. We want the asteroids to fall from different positions for a greater challenge!

So we're going to add some randomness to our game by changing where we spawn our asteroids. To do this we're going to make use of something called a library.

A library is like a collection of pre-made functions for us to use. For our case, we're going to use the random library to access a special kind of function called random.randint.

random.randint takes two arguments, a minimum value and a maximum value. Here's an example!

```
random.randint(-640, 640)
```

When we call this function, the return value is some random number that is between the minimum and maximum number. For our example, we put -640 as the minimum value and 640 as the maximum value. That means the random number will not be lower than -640 and will not be higher than 640.

Now if we make the x coordinate of our asteroid return a random value, it will start falling from a different x position everytime. Let's do that now.

Game Loop

```
import random

self.asteroidTimer = self.asteroidTimer + 1

if self.asteroidTimer >= 120:
    self.asteroidTimer = 0
    self.newAsteroid = Hazard()
    self.newAsteroid.x = random.randint(-640, 640)
    self.newAsteroid.y = 400
```

Here,we import the random library to our game. It comes packaged with a bunch of different functions related to random generation, but here we use just one particular function called "random.randint".

By adding this code, we allow the asteroid to spawn between x values of -640 to 640. Because we are not changing the y position, it will always fall from the top of our screen.

SAVE + PLAY

Asteroids will now randomly spawn across the top of the screen!

05.
CHAPTER

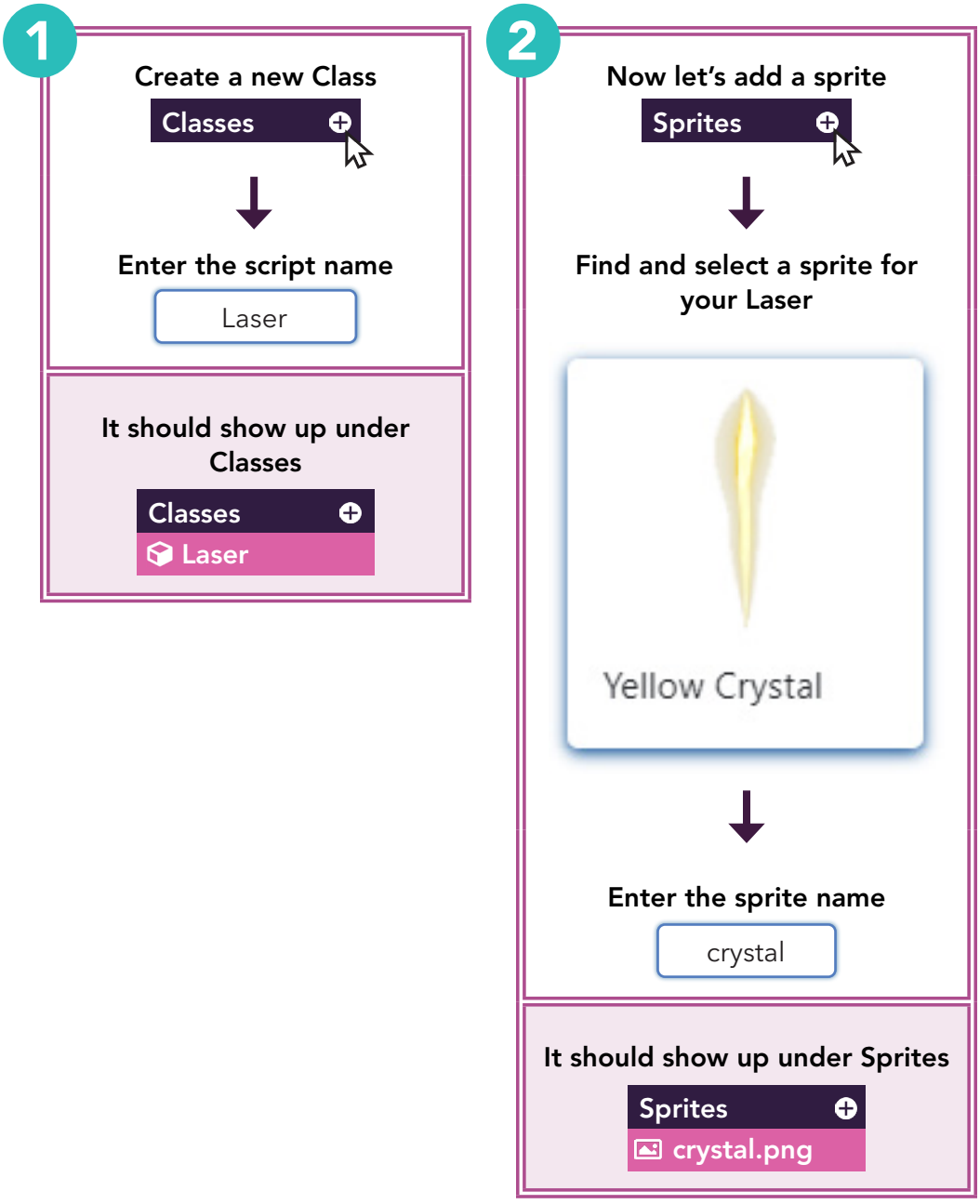
SHOOTING LASERS

Right now the only way we can avoid asteroids is by flying around them. To make the game more fun, we're going to add the ability to shoot laser beams!

To do this we're going to draw on many concepts we've learned already:

- Creating a class
- Uploading and assigning a sprite to an object
- Checking if a key is pressed
- Spawning an object
- Editing that object's Class to make it move within the screen consistently

Alright! So our first step will be to create a Laser class and then editing the code in the Player class to check if the player has pressed the J key:



Player | Loop

```
if key_is_pressed("D"):
  self.x = self.x + 2

if key_is_pressed("A"):
  self.x = self.x - 2

if key_is_pressed("W"):
  self.y = self.y + 2

if key_is_pressed("S"):
  self.y = self.y - 2

if key_was_pressed("J"):
  self.crystal = Laser()
  self.crystal.x = self.x
  self.crystal.y = self.y

asteroidHit = get_collision(self, "Hazard")
if asteroidHit:
    ...
...
```

You may have noticed that this time we check if **key_was_pressed** instead of **key_is_pressed**.

The difference this time is that we only want to spawn a laser if we press the key once and only once.

Key_was_pressed checks if the key was pressed only on the current frame of the game, that way we only shoot one laser beam instead of continuously checking if the key is being held down.

If we press the J key on this frame, we create a new laser beam with Laser() and set its position to our spaceship's position using self.x and self.y.

SAVE + PLAY

Right! Looks like we haven't assigned a sprite to our Laser yet, let's go do that in our Laser class.

Laser | Start

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

This line simply assigns the crystal sprite we added to the laser object.

SAVE + PLAY

Oh no! Our laser beams don't move at all when you press the J key. Don't worry, that's because we haven't added movement yet to the lasers.

Let's fix this by adding a line of code that will make the crystal constantly move upwards.

Laser | Loop

```
self.y = self.y  + 4
```

Here, we update the crystal Laser's y value by 4 inside the loop tab.

SAVE + PLAY

Our lasers now move upwards toward the top of the screen. However, when they touch an asteroid, nothing happens.

We want both the asteroid and laser to get destroyed when they collide with each other.

Laser | Loop

```
self.y = self.y  + 4

asteroidHit = get_collision(self, “Hazard”)
if asteroidHit:
  destroy(asteroidHit)
  destroy(self)
```

The if statement here checks for collisions between ourselves and an asteroid, and if a collision is detected, destroy the asteroid and the laser, clearing the way for our player.

SAVE + PLAY

You should now see your Lasers destroy each asteroid and itself as they collide.

06.
CHAPTER

HEALTH

Right now we die instantly with 1 hit, but we want to be able to keep playing for a bit when we hit asteroids. This means we need to add health!

First we need to create a health count, then we need to make it so that every time the player hits the asteroid, our health goes down by 1. So let's add a health counter that starts at 5, this means we'll have 5 lives.

Player Start

```
self.sprite = sprite("ship.png")
self.health = 5
```

Here, we create a variable called health and assign it with the value 5.

Next, we need to subtract 1 from this health count every time we hit an asteroid.

Scroll to your **get_collision** between the Player (self) and the Hazard (asteroid).

Player Loop

```
...

if key_was_pressed("J"):
  self.crystal = Laser()
  self.crystal.x = self.x
  self.crystal.y = self.y

asteroidHit = get_collision(self, "Hazard")
if asteroidHit:
   destroy(asteroidHit)
  ~~destroy(self)~~
   self.health = self.health - 1
   print("Health: " + str(self.health))
```

Instead of destroying the player immediately, we want to decrease our health every time we hit an asteroid, so we remove the "destroy(self)" and replace it with "self.health = self.health - 1".

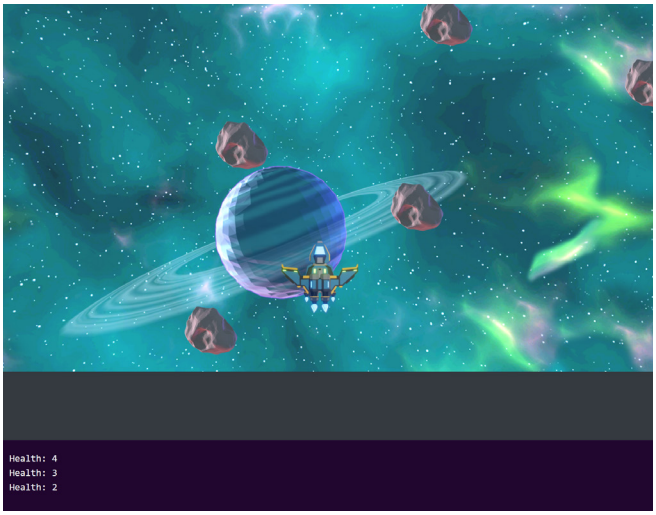
Below that, we take the current health variable we created and subtract 1 from it.

Finally, we print out the health our player has to the console each time it collides. In plain English, the print statement says: Print the word "Health: " and then combine the number value of the health variable to the end.

This way, you will always know what the health value is after you collide with the asteroid.

Notice the str() function. This function converts data, such as a number, into a string (text). A string is a term in coding for text. If we don't wrap the self.health inside the str() function, then PixelPad would give us an error in the console. This is because we would be trying to combine two different things together, in this case, combining a string (text) with a number.

SAVE + PLAY



Notice how we can see the number value of the Player's health in the console. This is thanks to the print statement that we just put inside the get_collision function in the Player class.

Now that we have a health counter, we want to be able to destroy our player once our health reaches zero. To do this, we're going to write an if statement inside our Player Loop that checks to see if the Player's health has dropped below 1. Once it successfully checks that the health is 0, we can tell it to destroy itself.

Player | Loop

```
...

if key_was_pressed("J"):
  self.crystal = Laser()
  self.crystal.x = self.x
  self.crystal.y = self.y

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

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

Here, we've added an if statement saying: If the player's health is less than or equal to 0, then destroy the Player.

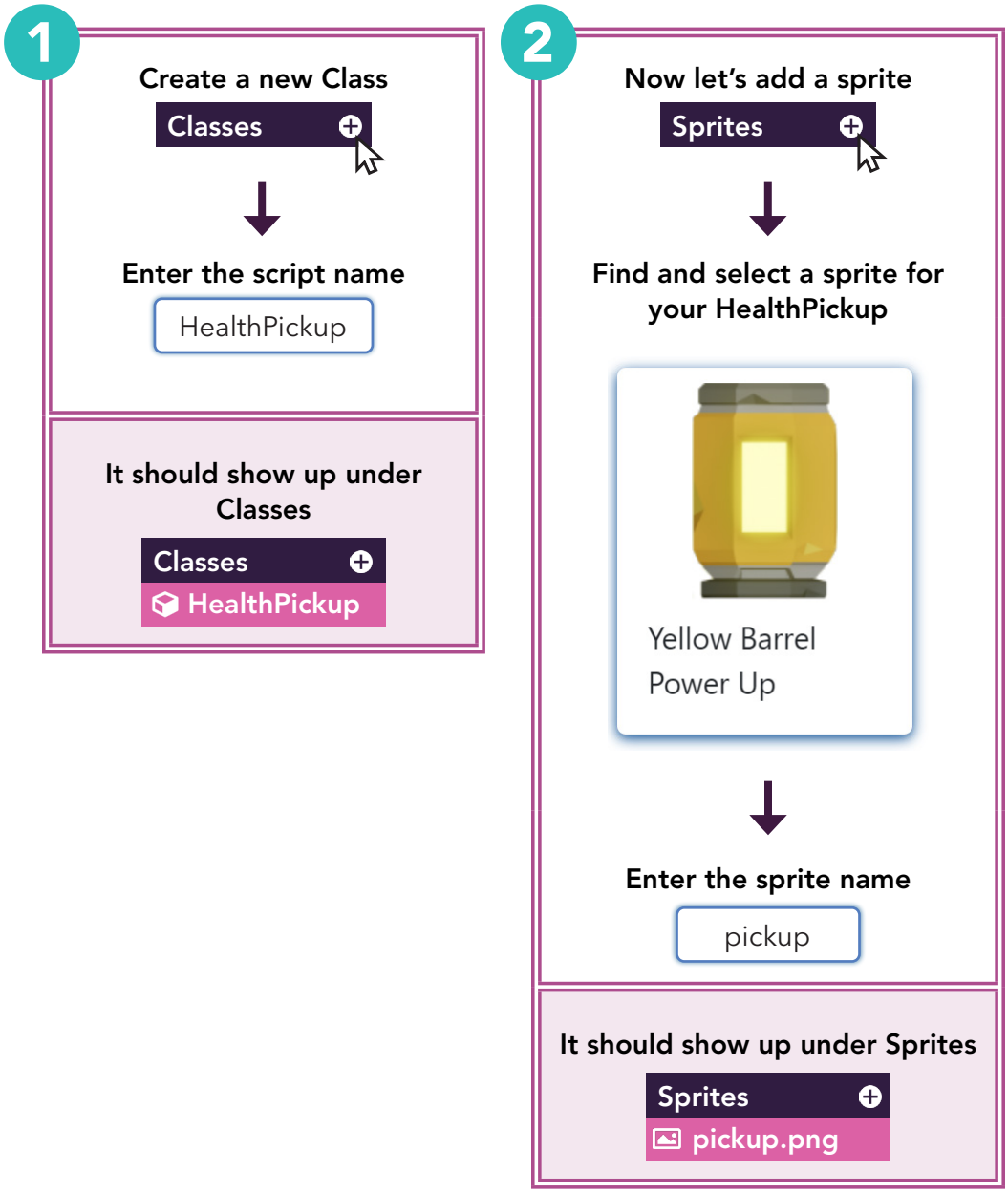
SAVE + PLAY

You should now be able to hit 5 asteroids before your ship is destroyed!

HEALTH PICKUPS

Now that our Player can lose health and die, we need some way to regain lost health to survive! That means we need to make health pickups. To implement this feature, we're going to use multiple concepts we've learned already, including collision checking, timers, and variable manipulation.

First, similar to what we have done before, we'll start by creating a HealthPickup class and uploading a sprite from the asset library so we can assign it to the object.



HealthPickup Start

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

self.scaleX = 0.5
self.scaleY = 0.5
```

Here, we're assigning the sprite "pickup.png" to the HealthPickup sprite variable.

Lastly, we use scaleX and scaleY to decrease the size of the sprite horizontally and vertically.

Feel free to play around with the scaleX and scaleY values and see what happens!

Now we will add a timer for our health pickups to create a spawner for them.

Game Start

```
self.space = Background()

self.hero = Player()
self.hero.x = 0
self.hero.y = -200

self.asteroid = Hazard()
self.asteroid.x = -10
self.asteroid.y = 250

self.asteroidTimer = 0

self.healthPickupTimer = 0
```

This line just creates a variable to store how much time has passed since the last healthPickup was spawned. It starts at 0 just like asteroidTimer.

Now that we have our timer, it's time to create the loop condition to spawn our pickups.

Game | Loop

```
self.asteroidTimer = self.asteroidTimer + 1

if self.asteroidTimer >= 120:
  self.asteroidTimer = 0
  self.newAsteroid = Hazard()
  self.newAsteroid.x = random.randint(-640, 640)
  self.newAsteroid.y = 400

self.healthPickupTimer = self.healthPickupTimer + 1

if self.healthPickupTimer >= 300:
   self.healthPickupTimer = 0
   self.healthShield = HealthPickup()
   self.healthShield.x = random.randint(-640, 640)
   self.healthShield.y = 360
```

This code is very similar to the asteroid timer above.

On every frame, the health pickup timer goes up by one. When healthPickupTimer reaches 300 a new HealthPickup will be spawned. (This is 5 seconds, 5 x 60 frames = 300 frames)

SAVE + PLAY

Notice how the HealthPickups spawn but they stay in one place? Similar to what we did with the Player's Lasers, we can fix this by adding code to the HealthPickup class to make it move downwards toward the bottom of the screen!

HealthPickup | Loop

```
self.y = self.y - 2
```

This line simply makes the health pickup move downward every frame. Feel free to change the speed of the HealthPickup by changing the number value.

SAVE + PLAY

Great! Our health pickups now spawn and move toward the bottom of the screen!

Next, we'll need to make these pickups actually work. That is, the Player's health should increase by 1 every time a HealthPickup collides with the Player.

Player | Loop

```
...

asteroidHit = get_collision(self, "Hazard")
if asteroidHit:
  destroy(self.asteroidHit)
  self.health = self.health - 1
  print("Health: " + self.health)

healthShieldHit = get_collision(self, "HealthPickup")
if healthShieldHit:
  destroy(healthShieldHit)
  self.health = self.health + 1
  print("Gained Health!")
  print("Health: " + str(self.health))

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

First, we check if we're colliding with a HealthPickup with the function "get_collision()".

Then, we assign that collision object to a variable "healthShieldhit" so we can refer to it.

Using that variable reference we created, we can destroy the pickup object we collided with.

Next, we get our current health and add 1 to the current value of health.

Finally, we print ("Gained Health!") along with the current value of health so we know it worked and added health.

SAVE + PLAY

You should now see a message that tells us when the Player's health has increased by 1 every time the Player collides with a HealthPickup object.

```
Health: 4
Health: 3
Gained Health!
```

07.
CHAPTER

In this final chapter, we'll be using what we learned to add dangerous enemy ships that are going to fly towards us and shoot their own lasers at us!

SPAWNING OUR ENEMIES

To be able to spawn enemies, we will need to first create the enemy itself!

As usual, the first step will be to create an Enemy class and then find a sprite in the asset library to assign it to our newly created enemy.

1

Create a new Class

Classes ⊕

↓

Enter the script name

Enemy

It should show up under Classes

Classes ⊕

Enemy

2

Now let's add a sprite

Sprites ⊕

↓

Find and select a sprite for your Enemy

↓

Enter the sprite name

redShip

It should show up under Sprites

Sprites ⊕

redShip.png

Enemy Start

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

Here, we assign "redShip.png" to our Enemy sprite variable.

Now that we have a sprite assigned, let's make sure that our enemy can move downwards.

Enemy Loop

```
self.y = self.y - 1
```

In this one line we make the enemy move downwards by -1 on the y axis every frame.

SAVE + PLAY

You might notice the enemy doesn't do anything or even appear on the screen. This is because we haven't done anything to spawn it yet!

Let's go ahead and create a spawner. Our first step is to create a timer in the Game class.

Game Start

```
self.space = Background()

self.hero = Player()
self.hero.x = 0
self.hero.y = -200

self.asteroid = Hazard()
self.asteroid.y = 300

self.asteroidTimer = 0

self.healthPickupTimer = 0

self.enemyTimer = 0
```

Here we simply add a timer for spawning enemies.

Next, we need to continuously add 1 to the enemyTimer so that it counts up, much like a stopwatch! Then we can use this timer to spawn enemies when this timer reaches a certain number of seconds.

Game | Loop

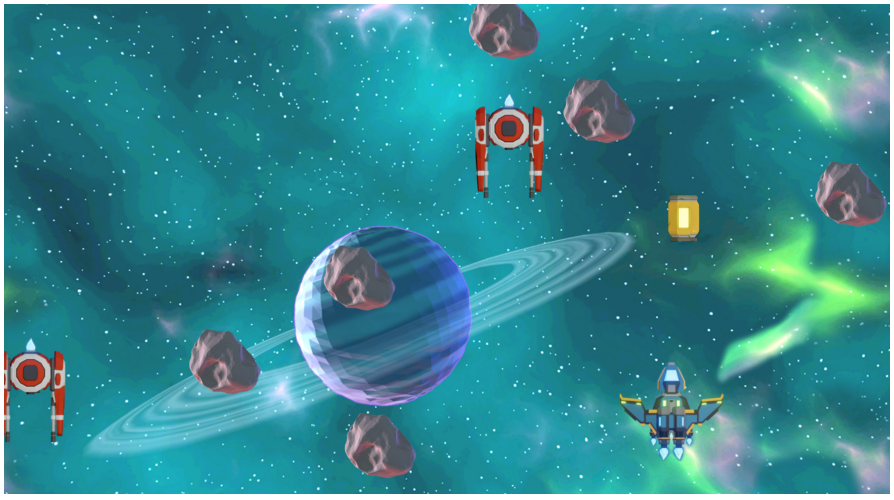
```
...

self.healthPickupTimer = self.healthPickupTimer + 1

if self.healthPickupTimer >= 300:
   self.healthPickupTimer = 0
   self.healthShield = HealthPickup()
   self.healthShield.x = random.randint(-640, 640)
   self.healthShield.y = 360

self.enemyTimer = self.enemyTimer + 1

if self.enemyTimer >= 350:
   self.enemyShip = Enemy()
   self.enemyShip.x = random.randint(-640,640)
   self.enemyShip.y = 360
   self.enemyTimer = 0
```

Here, we add 1 to the enemy timer every frame, then we have an if statement to check the value of the timer.

If the timer is greater than 350, we spawn a new enemy at a random position along the x axis between -640 and 640, and at a y position of 360 (top of the screen).

After all of that we reset the timer back to 0 so that the timer can continue to count up from 0 again to repeat the cycle.

SAVE + PLAY

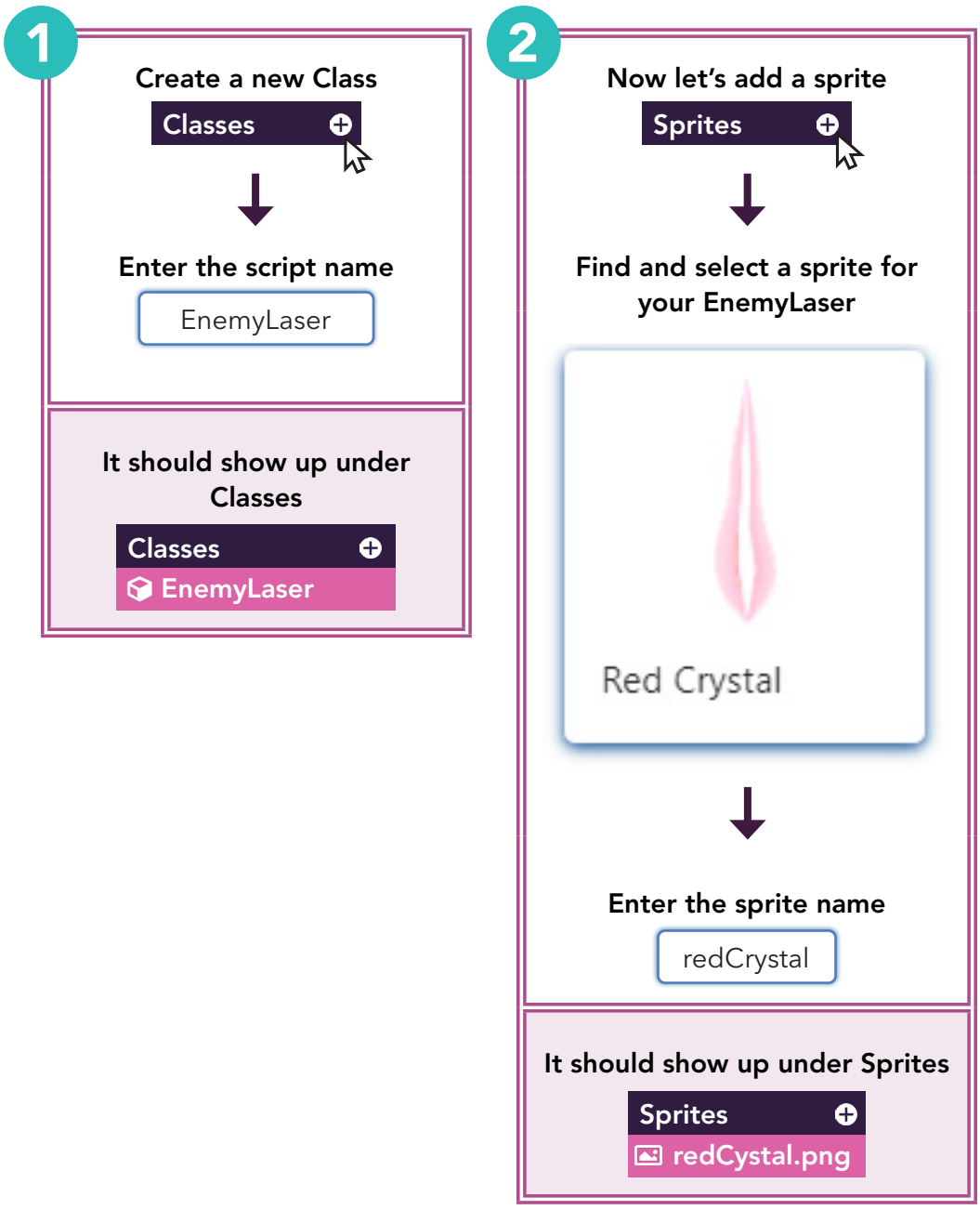


Not only do our enemies move downwards from the top of the screen now, but they also spawn randomly horizontally (along the x-axis).

Next, we'll make the enemies fire lasers as they fly down!

CREATING AND SPAWNING ENEMY LASERS

The first step is, you guessed it! To create an Enemy Laser class, then find and assign a sprite to it from the asset library.



EnemyLaser | Start

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

Here, we assign the sprite "redCrystal.png" to our EnemyLaser sprite variable. (The name of the sprite does not have to match the one shown above. It depends on the sprite you've uploaded).

Now that we have a sprite for our enemy's laser, let's make sure they move downwards continuously on our screen.

EnemyLaser | Loop

```
self.y = self.y - 2
```

Just like our asteroids and health pickups move downwards, we can use the same code to make the enemy laser move downward.

Here, we're simply editing the y position of the laser to constantly move downward by 2.

Next, we'll need to create a timer that our Enemy can use to fire its laser.

Enemy | Start

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

self.shootTimer = 0
```

Here, we create a variable "shootTimer". We will use this timer to control when the enemy shoots its Lasers.

Now that the timer is set up, let's spawn EnemyLasers using the timer we've just created!

Enemy | Loop

```
self.y = self.y -1

self.shootTimer = self.shootTimer + 1

if self.shootTimer >= 120:
  self.redCrystal = EnemyLaser()
  self.redCrystal.x = self.x
  self.redCrystal.y = self.y
  self.shootTimer = 0
```

Here, we add 1 to shootTimer on every frame.

After that, we create an if condition that if the timer becomes larger than or equal to 120, then create a new object called redCrystal of the type EnemyLaser, make it spawn at the location of the enemy, and reset the shootTimer back to 0. (120 is 2 seconds)

SAVE + PLAY

Awesome! Our enemies can now shoot lasers as they move down! Our game is looking great so far!

Now that we have a functioning shootTimer, we will first, check when an Enemy ship hits a Player and second, check when a Player's laser hits an Enemy.

Enemy | Loop

```
self.y = self.y -1

self.shootTimer = self.shootTimer + 1

if self.shootTimer >= 120:
  self.redCrystal = EnemyLaser()
  self.redCrystal.x = self.x
  self.redCrystal.y = self.y
  self.shootTimer = 0

playerHit = get_collision(self, "Player")
if playerHit:
    playerHit.health = playerHit.health - 1
    destroy(self)

laserHit = get_collision(self, "Laser")
if laserHit:
    destroy(laserHit)
    destroy(self)
```

This code should look familiar! Here, we're using the get_collision() function to check for a collision between an Enemy object and the Player. When such a collision happens, we decrease the Player's health by 1 and destroy the Enemy.

Lastly, we use another get_collision() function to check for a collision between an Enemy and a Player's laser. When such a collision happens, we destroy both the Enemy ship and the Player's laser.

SAVE + PLAY

Now our Enemies and Player's Lasers should disappear when they collide with one another.

Next, we're going to add another get_collision() function to the Player so that every time we hit an Enemy laser, we're going to subtract 1 from our health.

Player | Loop

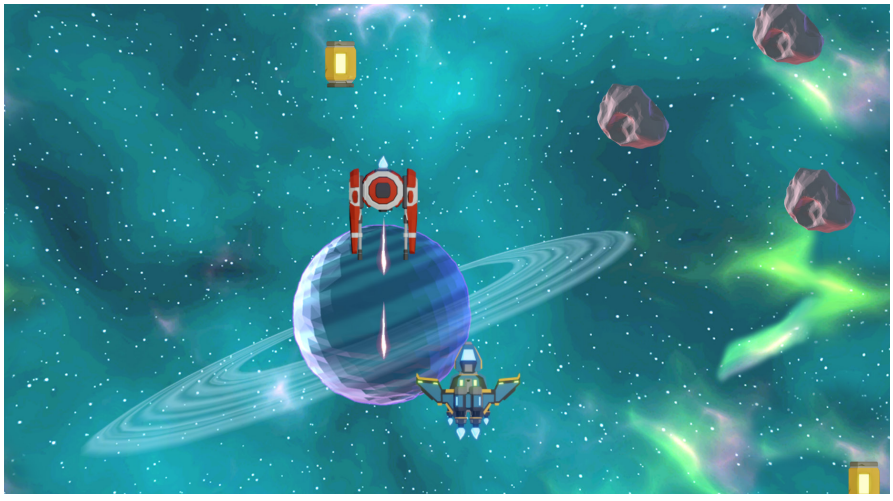
```
...

healthShieldHit = get_collision(self, "HealthPickup")
if healthShieldHit:
  destroy(healthShieldHit)
  self.health = self.health + 1
  print("Gained Health!")
  print("Health: " + str(self.health))

laserHit = get_collision(self, "EnemyLaser")
if laserHit :
  destroy(laserHit)
  self.health = self.health - 1

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

Here, we've added a get_collision between the Player and an Enemy Laser.

When such as collision happens, we'll, again, subtract 1 from the Player's health and destroy the Enemy Laser.

SAVE + PLAY

Now that we've added this code, when the Player collides with Enemy Lasers, our health will decrease by 1!



You should now notice that enemies spawn and shoot lasers towards you!

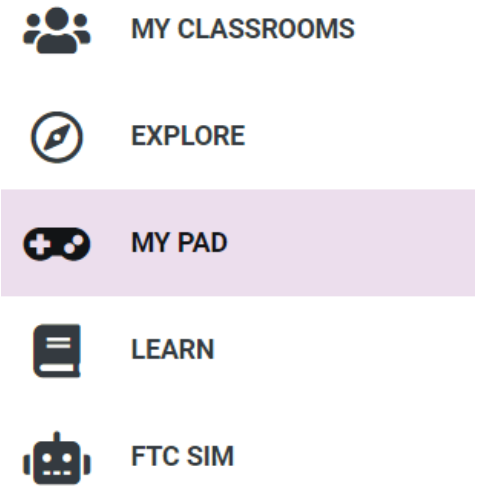
If your game functions properly, congratulations, you've successfully completed PY101!

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!

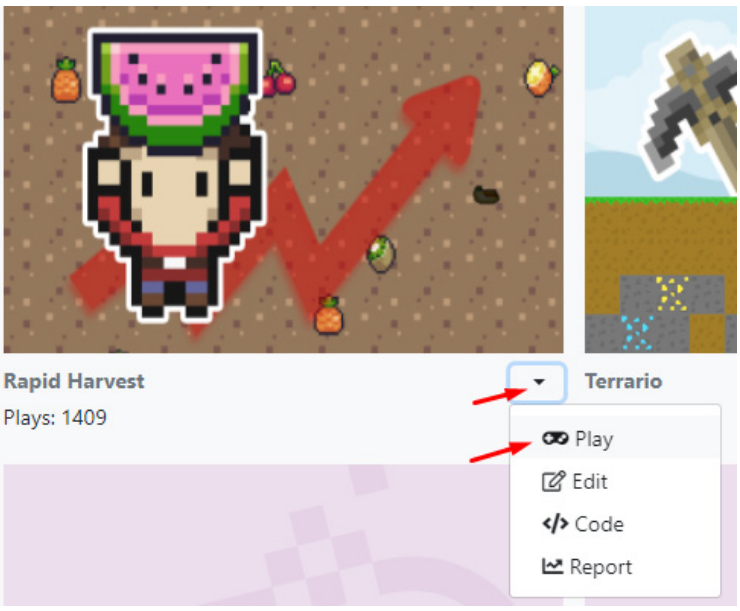
SHARING MY GAME

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

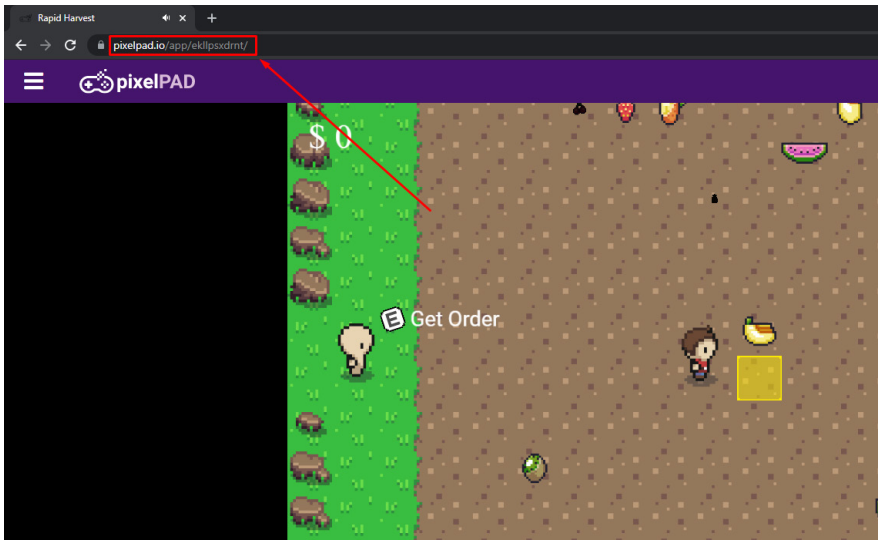
1. Go to the MyPad Section



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

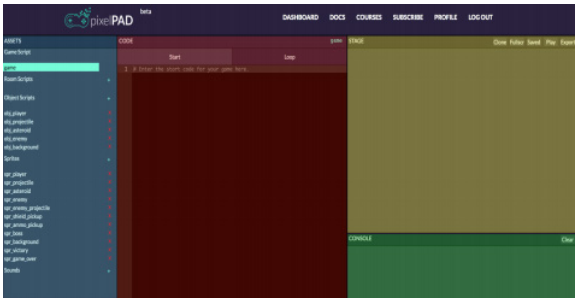
The following activities are optional and should be added to your current game. Most of them can be added during your game's development, but some might require your game to be already completed. You can check the prerequisite chapters beside the activities to know if you are able or not to do it at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

#	Prerequisite	Activity
1	Chapter 3	Create two new types of asteroids: › One that moves from left to right › One that moves from right to left
2	Chapter 5	Create spawners for the two other asteroids added in the activity above.
3	Chapter 6	Add randomness to your asteroids' speed so that each asteroid will have a different speed.
4	Chapter 8	Add a bad pickup that will take away 2 health points from the player.
5	Chapter 9	Add a second power up to your game: › This power up should allow the player to shoot 3 bullets at once for 5 seconds

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!

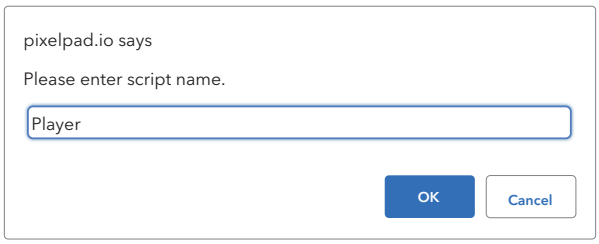
CONSOLE: Your console is where you will see messages when there are errors in your code, and also where you can have messages from your game show up such as the score, or instructions on how to play your game.

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"



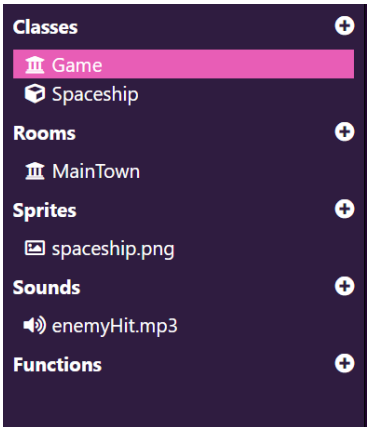
Then type in any name you'd like. My particular convention looks like this:

- "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 Player class, saying "self" refers to the Player class itself.

Examples of Self

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

The code would make Player 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:

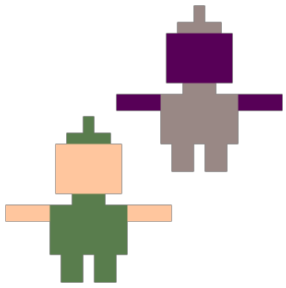
```
if get_collision(self, 'Asteroid'):
    print('The spaceship has collided with an asteroid')
```

- 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.

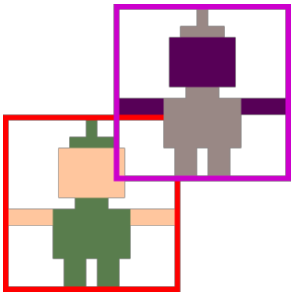
BOUNDING BOXES

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.

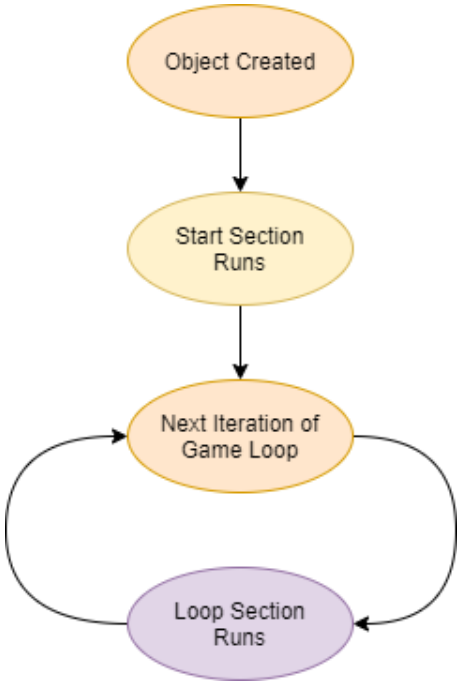
The Game Loop

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

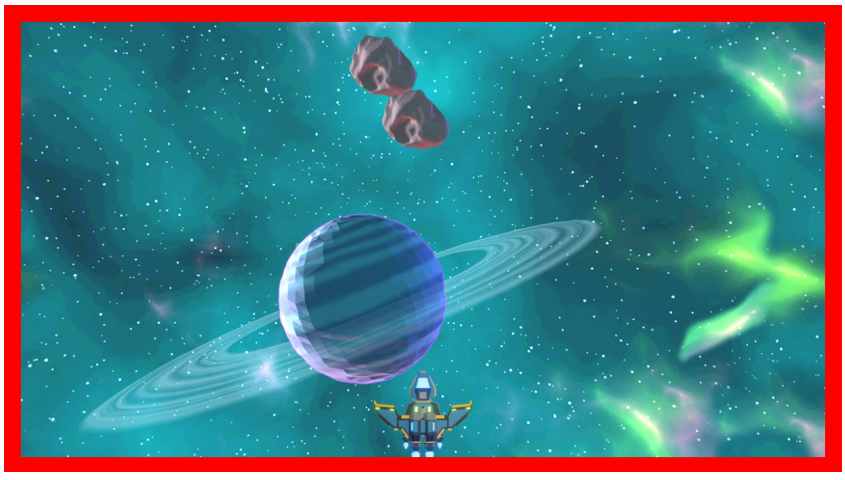
If we have code like: x > 300, this is a specific kind of condition called a comparison. Comparisons are true/false questions we can ask the computer about pairs of numbers. There are six main kinds of comparisons, each with its own operator (special symbol). This table shows an example of each kind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

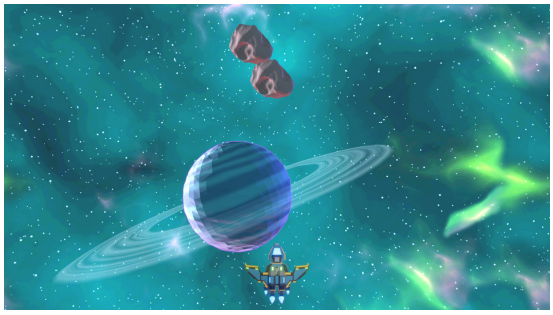
Every frame in our game lasts exactly the same amount of time: 1/60th of a second. That means that there are 60 frames in a second.

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.

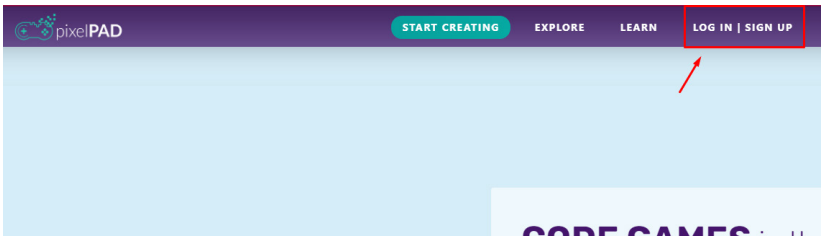
2. Next, we Read our code. Using what we learned from running our code, we look for specific commands that might be causing problems. Sometimes, an error message will tell us exactly where to look by giving us a line number and file name (for example, error in Player.Loop() on line 4 means that the 4th 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 LEARN and My PAD section on PixelPAD?

_____

_____

_____

CHAPTER 2

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

player = _____()

CHAPTER 3

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 4

In the following code, between what two objects is a collision being checked for?

```
if get_collision("Ally", "Spikes"):
   destroy(self)
```

a) self and **Spikes**

b) self and **Ally**

c) **Ally** and **Spikes**

CHAPTER 5

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

a) One Minute?

_____

b) One Hour?

_____

Bonus: One Year?

_____

CHAPTER 6

What is the library that we import randint from?

```
import _______________
```

CHAPTER 7

What is the difference between key_is_pressed and key_was_pressed?

_____

_____

CHAPTER 8

What is the operator that allows you to print both a variable and words together?

Example: `print("Words" _____ variable)`

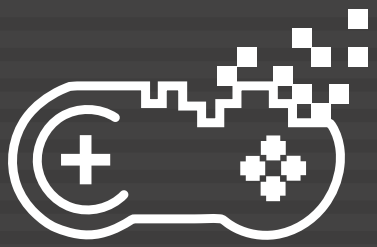
a) =

b) +

c) -

d) #

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 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 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:

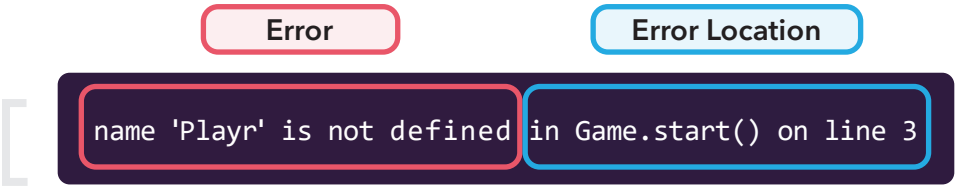
1. First, we Run our code. Running our code will let us observe it, which will show us whether there are any errors or other problems. If everything is working properly, we can stop debugging.

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.



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
```

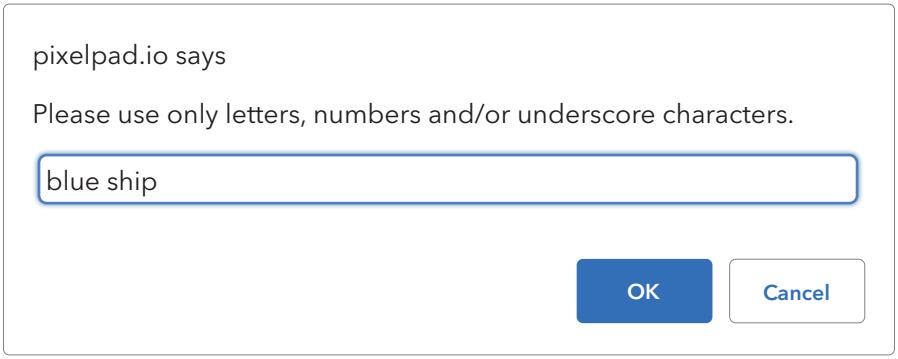
You're trying to access the variable X inside the Spaceship object. However, that variable doesn't exist.

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.