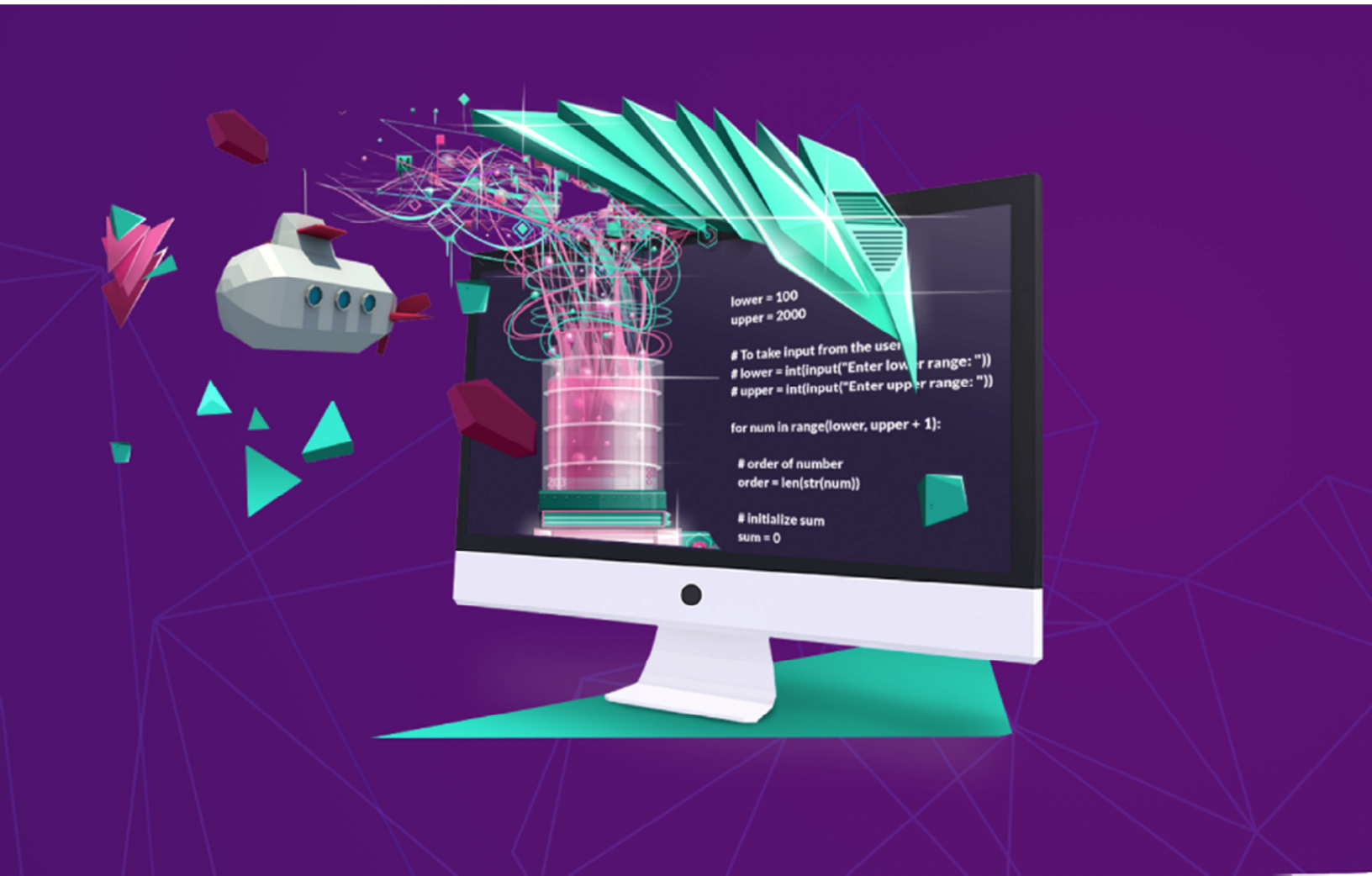
Python & PixelPAD

Introduction to App Dev with PixelPAD

FIRST Robotics Canada

Maker Labs Tutorial




FIRST®
CANADA

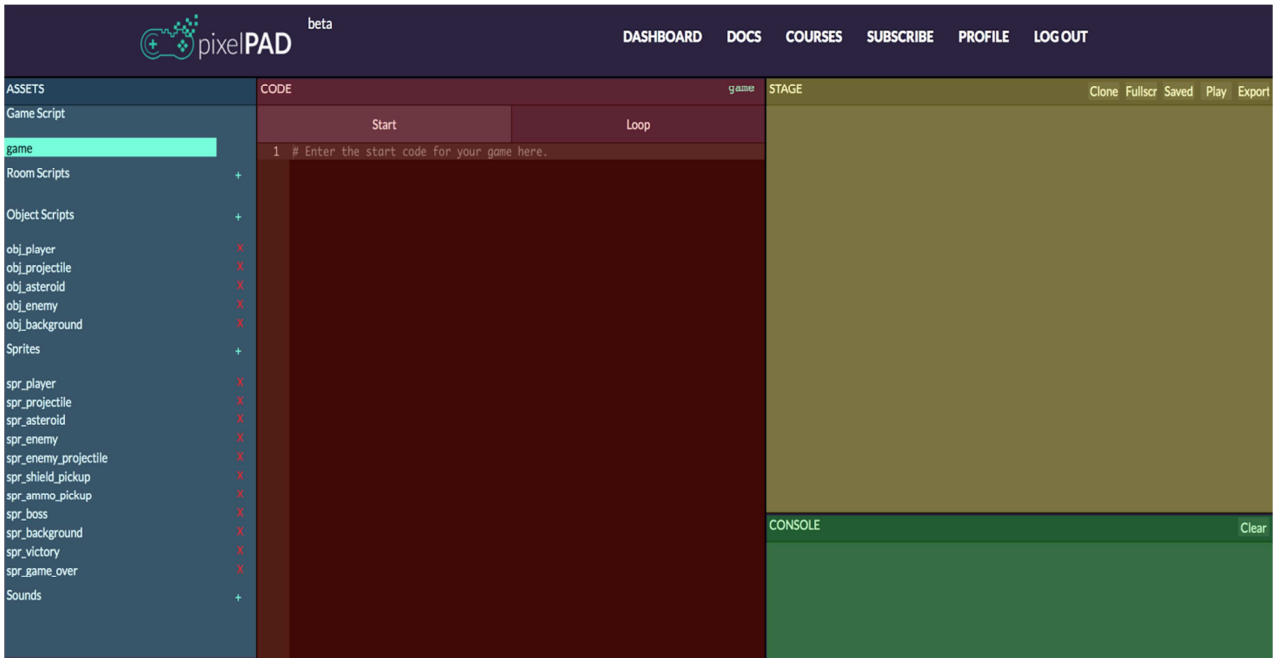

pixelPAD

Table of Contents

Getting Started

What is PixelPAD?

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

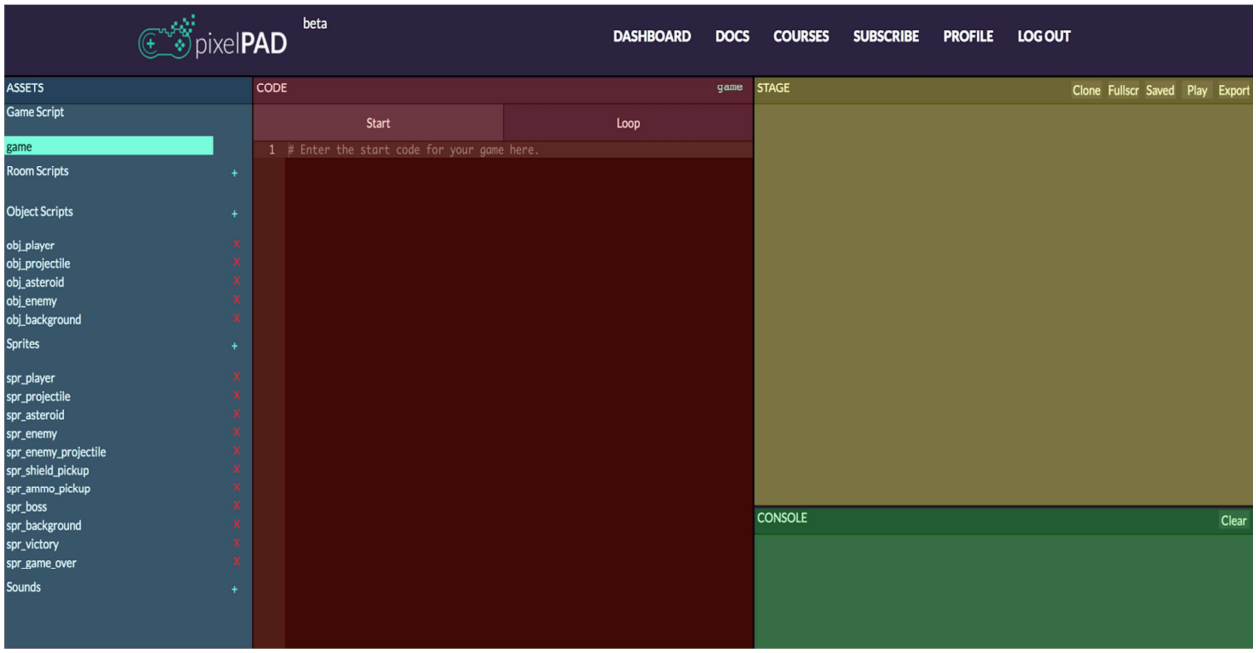


The PixelPAD IDE is composed of 4 areas:

ASSETS: Your assets are where you can add and access your object scripts and sprites. Scripts 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.

Getting Started



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

Two of the asset types we just covered, rooms and objects, 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 "room scripts", "object scripts" or "sprites"

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

"rm_openworld" -> room

"obj_player" -> player

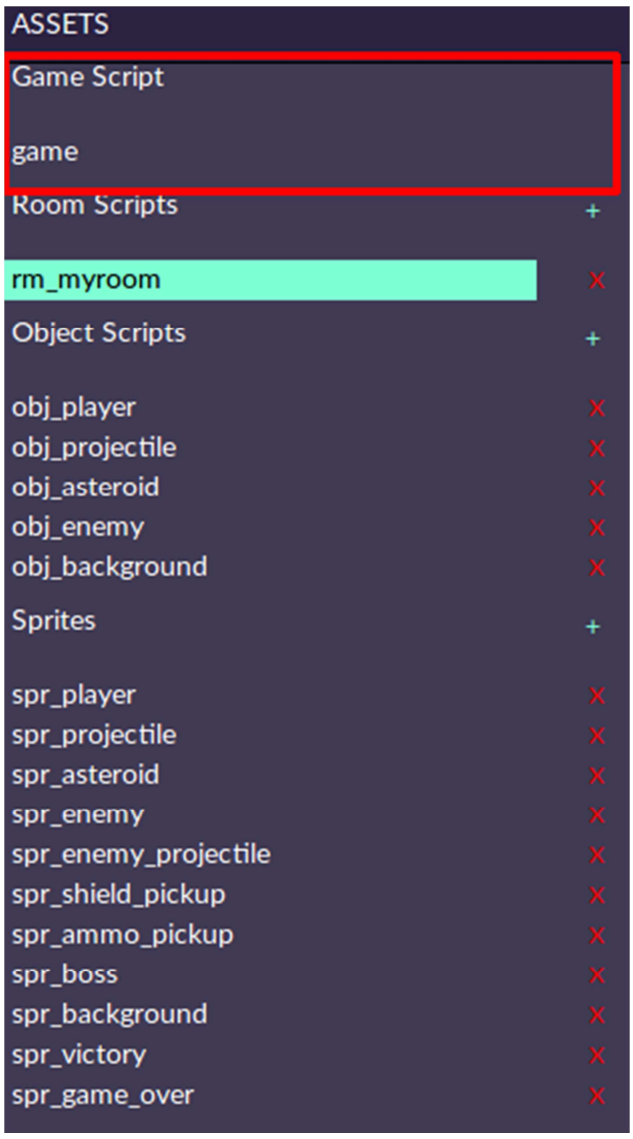
"spr_background" -> background

I always preface my assets with the type of asset followed by the name. This isn't necessary, but keeps your code neat and readable.

Scripts

The Game Script

There is one script asset that always exists in every project: the game script. The purpose of the game script is to load all of the other assets in our project. The game script represents our entire game, so it doesn't count as a room or as an object.



Default Object Properties

Sprite, sprite_height and sprite_width

Every object or object script inherits default properties when created in PixelPAD.

The First of these few properties you should learn about are:

.sprite, .sprite_height and .sprite_width.

.sprite is image of the object. The value of .sprite is an image object which we will get to later.

.sprite_height takes a float between 0 and 1 and stretches the sprite of the object lengthwise

.sprite_width 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 close 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 player1.x we're really saying "player1's x"

Examples of Dot Notation:

obj_player.x -> referse to obj_player's x value

obj_player.sprite_width -> referes to obj_player's image width (percentage)

The "Self" Property

Self refers to whichever object you are currently in.

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

Examples of Self
#Code inside "obj_FIRST_ball"

```
self.x = 50
self.y = 30
self.sprite_width = 0.5
self.sprite_height = 0.5
```

The code would make obj_FIRST_ball 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 to 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 collision_check function

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

Here is an example of a collision_check function:

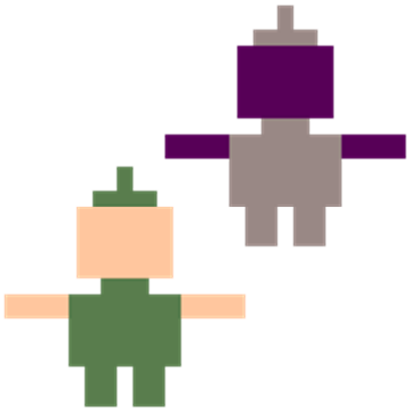
```
if collision_check(self, "obj_asteroid"):
  print("This object is colliding with an asteroid object")
```

- We start with an ordinary if statement.
- For our condition, we specify collision_check.
- 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 "obj_asteroid". This specifies the kind of object we want to check for collisions with. In this case, we are checking for collisions with obj_asteroid objects.

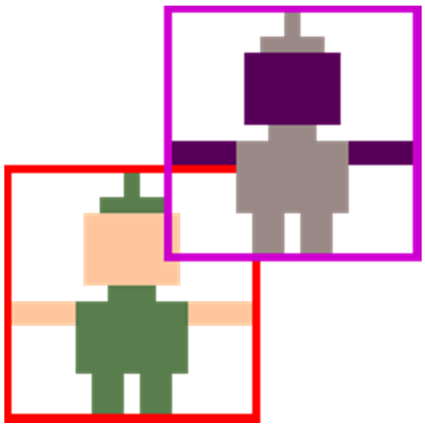
Bounding Boxes

Bounding Boxes

Every object has a bounding box, which is the rectangle that contains the object's entire sprite. The collision_check 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

Destroying Objects

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:

The Start and Loop

The Start and the Loop Tabs

Say you decide to go for a run. You put on your runners and then you run. Running the loop because it is repeats (one foot in front of the other), and putting runners on is 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 the robot. Whereas the "loop" could describe its animation.

The Game Loop

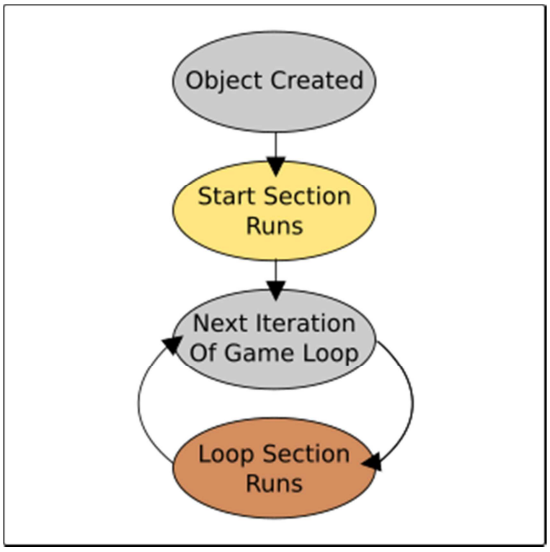
Loops exist in our day-to-day life. For example, you wake up, get ready, go to school, come back home, go to sleep and repeat these things every day! So looping is the act of repeating. In programming, loops describe instructions that repeat instead of having to code each instruction again and again!.

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 keeping jumping forever!

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

Loops

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 using object_new. 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

Conditionals

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:

- 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("left"), which is asking, "Is the left arrow 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 x = 300.

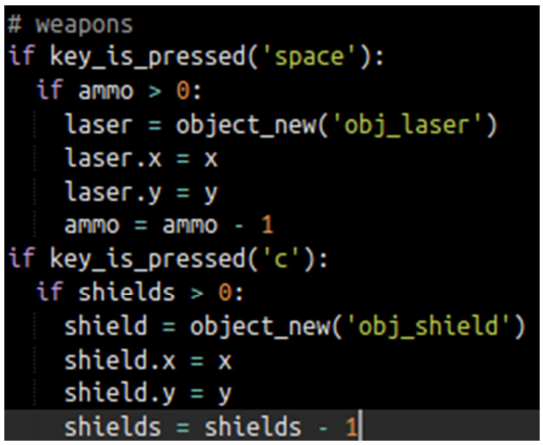
Indentation in Python

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 the our code. This helps us navigate our code and find bugs more easily.

For example:



```
# weapons
if key_is_pressed('space'):
  if ammo > 0:
    laser = object_new('obj_laser')
    laser.x = x
    laser.y = y
    ammo = ammo - 1
if key_is_pressed('c'):
  if shields > 0:
    shield = object_new('obj_shield')
    shield.x = x
    shield.y = y
    shields = shields - 1
```

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 quotation marks 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 left, right, up, and down.
- 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:

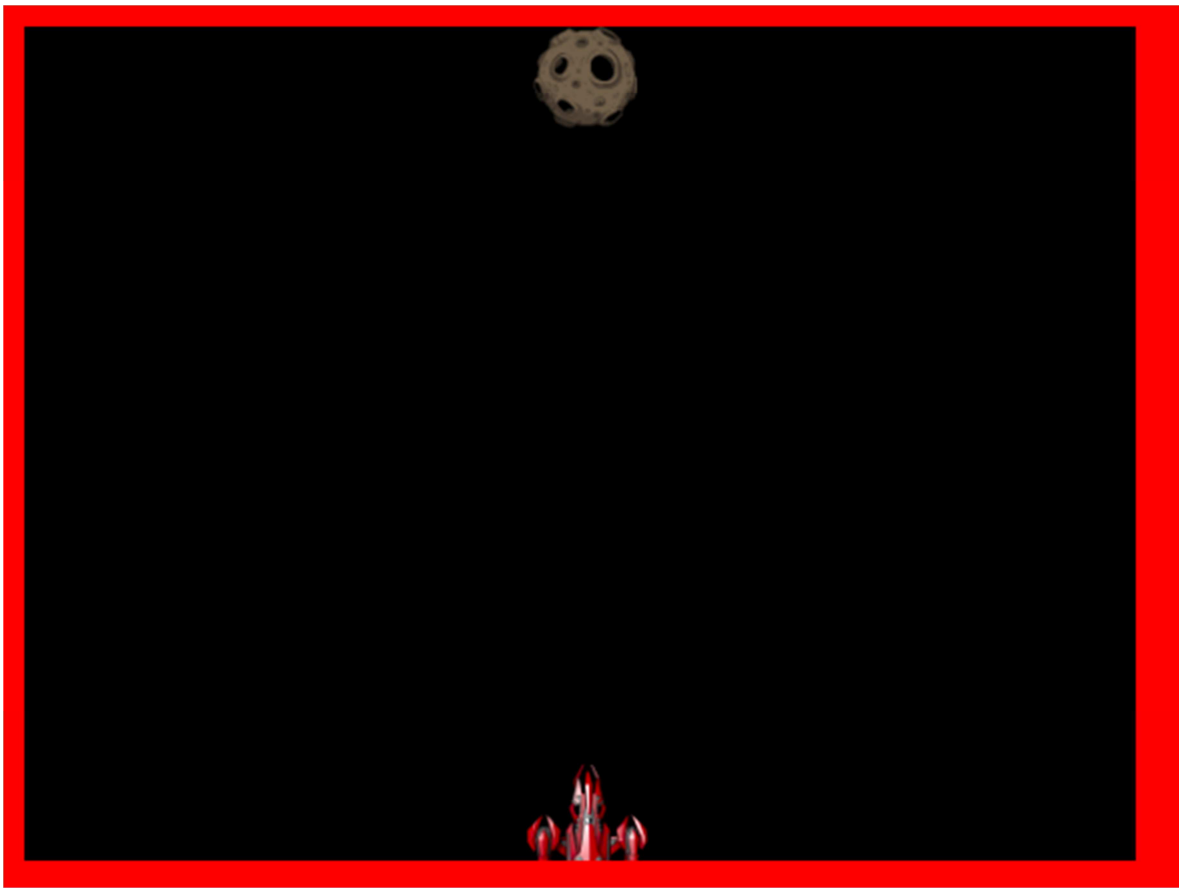
Example Question	Example Code
Is `x` smaller than `y`?	`x < y`
Is `x` bigger than `y`?	`x > y`
Is `x` smaller than or equal to `y`?	`x <= y`
Is `x` bigger than or equal to `y`?	`x >= y`
Is `x` equal to `y`?	`x == y`
Is `x` not equal to `y`?	`x != y`

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

Comparisons

Adding Boundaries Example

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



Comparisons

The Left Boundary

In our player script's Loop Section, add this new code at the bottom:

```
if x < -300:
    x = -300
```

The Right Boundary

Next, add this code:

```
if x > 300:
    x = 300
```

The Top Boundary

Next, add this code:

```
if y > 220:
    y = 220
```

The Bottom Boundary

Finally, add this code:

```
if y < -220:
    y = -220
```

Comments

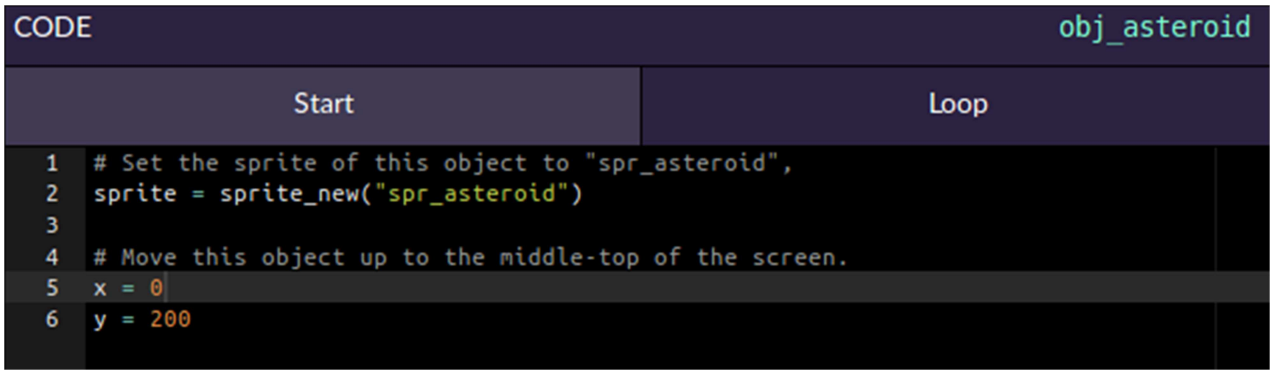
Explaining Code with Comments

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:



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.

Comments

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 makes the player move up when they press the "up" key
if key_is_pressed('up'):
  y = y - 10
```

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.

Comments

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:

```
# Add one to x
x = 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 very much thought, or because the author of the comment was told to comment their code. Here is an example of a vague comment:

Comments

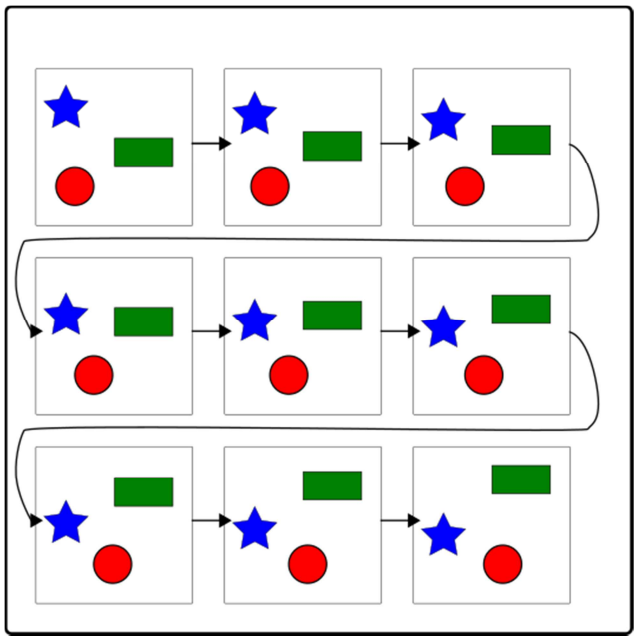
```
# weapons
if key_is_pressed('space'):
  if ammo > 0:
    laser = object_new('obj_laser')
    laser.x = x
    laser.y = y
    ammo = ammo - 1
if key_is_pressed('c'):
  if shields > 0:
    shield = object_new('obj_shield')
    shield.x = x
    shield.y = y
    shields = shields - 1
```

The comment at the top of this code just says "weapons." 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 weapons. 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. Here is an example of a movie with 9 frames:



Notice how each of the three objects is in a slightly different place every frame: the green rectangle moves up, the blue star moves down, and the red circle moves to the right. We can barely notice a difference between two consecutive frames, but the objects have moved quite a bit when you compare the first and last frames!

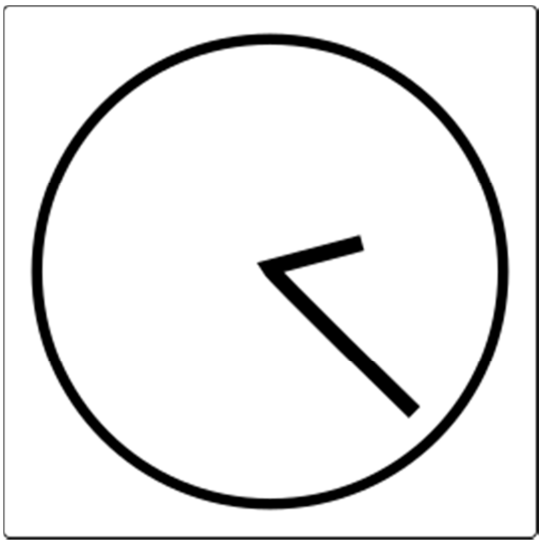
Frames Per Second

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

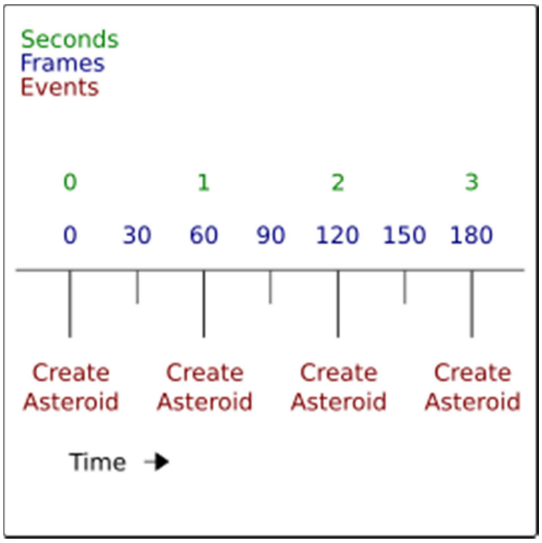
A timer is a number that counts time. For example, if we were watching a clock, and counted up by one every time the clock's second hand moved, we would be timing seconds.



Timers

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 = object_new("obj_asteroid")
asteroid.x = 0
asteroid.y = 200
```

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 = object_new("obj_asteroid")
asteroid.x = random.randint(-200, 200)
asteroid.y = 200
```

Random & Import

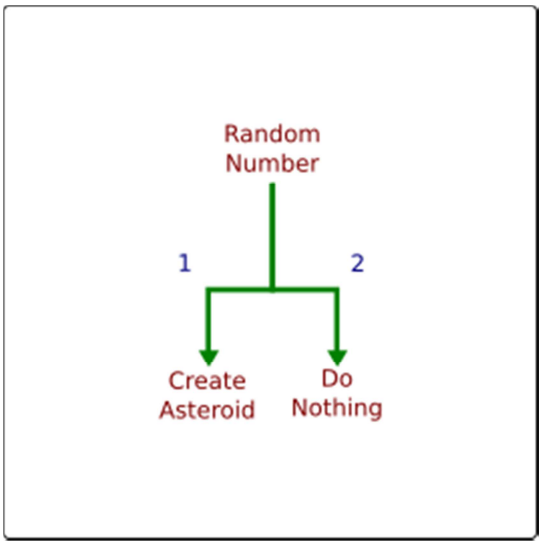
Random Function

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.



Random & Import

```
    if asteroid_frames >= asteroid_timer:
        asteroid_frames = 0
        if asteroid_count < asteroid_max:
            if random.randint(1, 2) == 1:
                new_asteroid = object_new("obj_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

When an object is created inside the `game` script, it is automatically made into a persistent object. 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.

Since persistent objects are part of the `game` script, they can be accessed in a special way: a persistent object called `foo` can be accessed from anywhere by writing `game.foo`.

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.

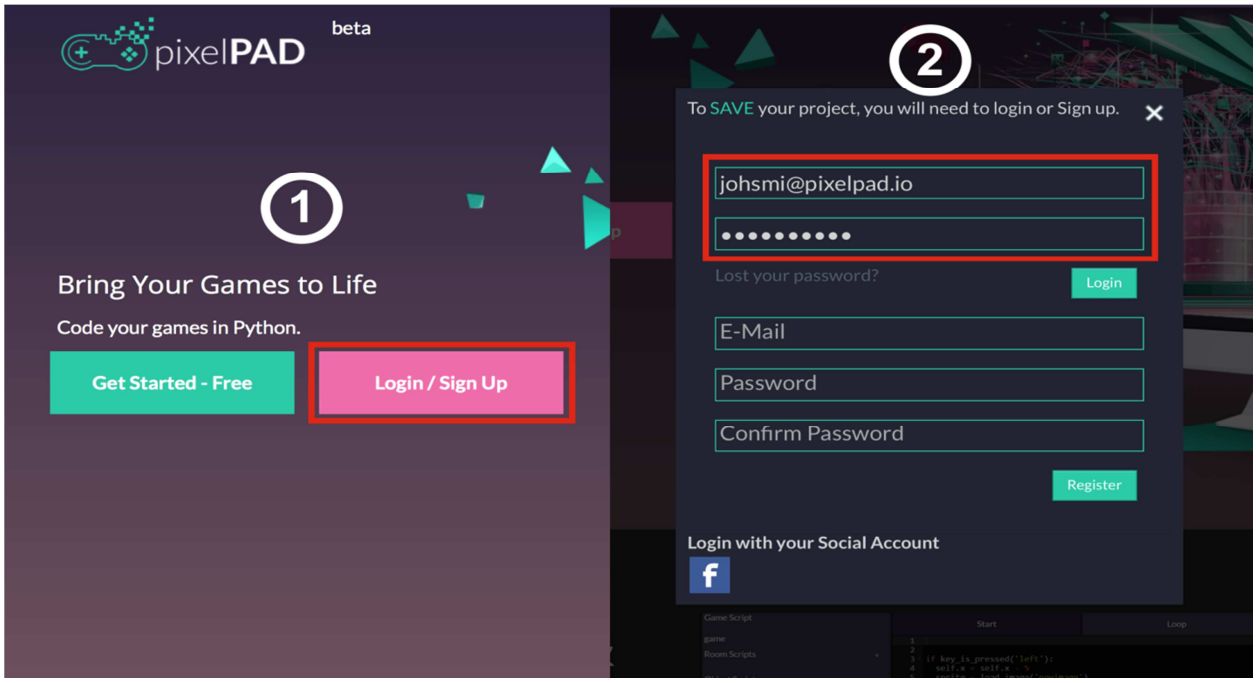
A better way to debug is to use the debug loop. The debug loop is a simple process that we repeat until our program works properly. This is what it looks like:

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 obj_player on line 4 means that the 4th line of code in the obj_player object 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 the Game Tutorial you'd like to work on and choose Asset Pack 1. This will download the necessary assets directly to your game to get you started.

Game Guide Preface

Preface - Read me

This tutorial tries its best to present all code in order. That is, code given to the user later in this tutorial should be placed after code already written into their program. For example, if we say:

a) In obj_player1's start tab write:

```
self.sprite_width = 0.5

self.sprite_height = 0.5
```

Then later on say:

b) In obj_player1 start tab write:

```
if self.name != "":

    print("hello " + self.name)
```

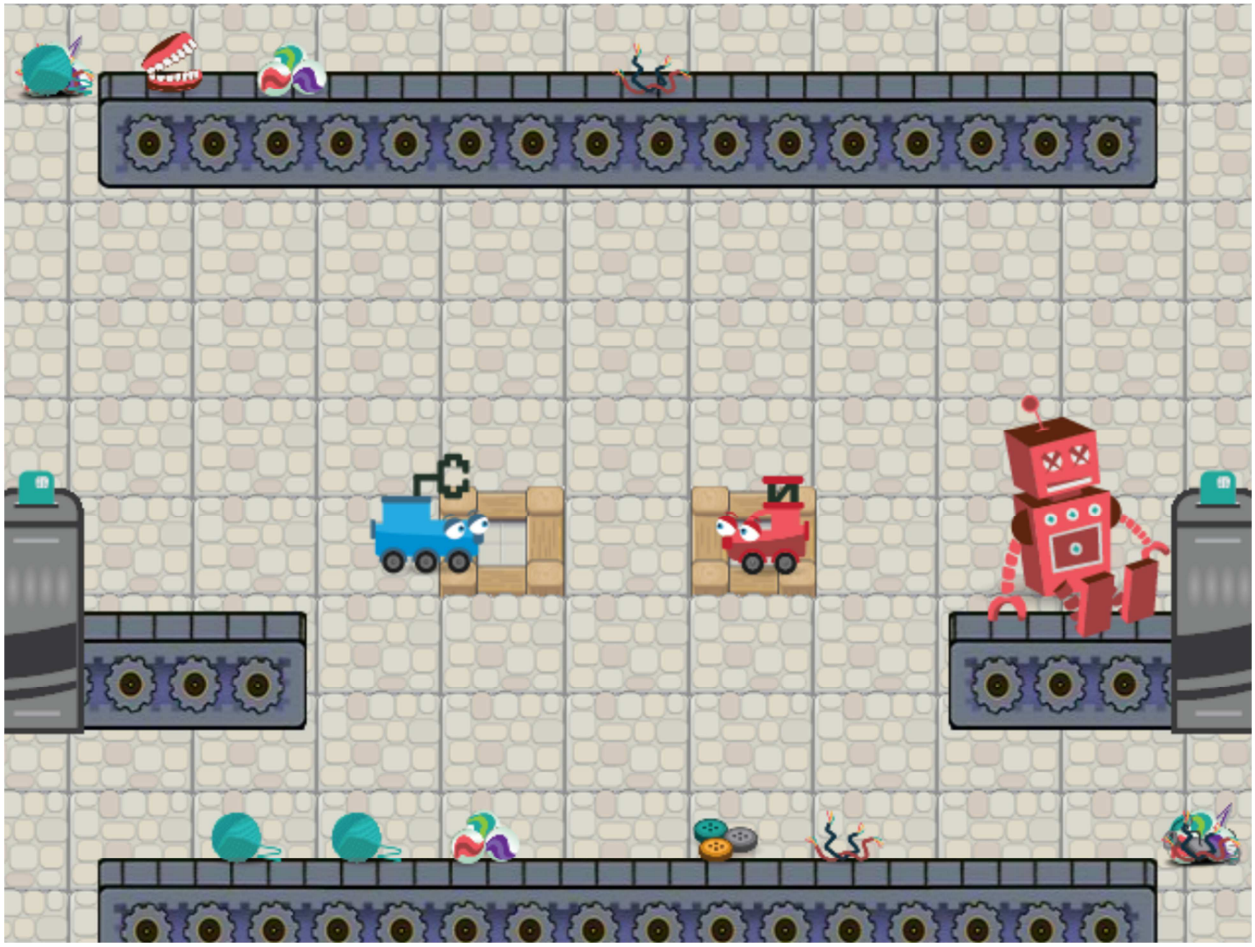
You can confidently place b)'s code after a)

Be cautious however, as from time to time we may ask the user to insert code within existing conditionals. In cases like this, read and understand the code carefully and place it in the most logical place.

There may also be times we ask the user to write their own code with the knowledge they've learned gained from the previous lessons. This code is necessary to ensure a fully functioning app.

Example Project

This is what your game will look like after you finish this tutorial:



You can try out an example project here:

https://pixelpad.io/play/32dwx2wr433/

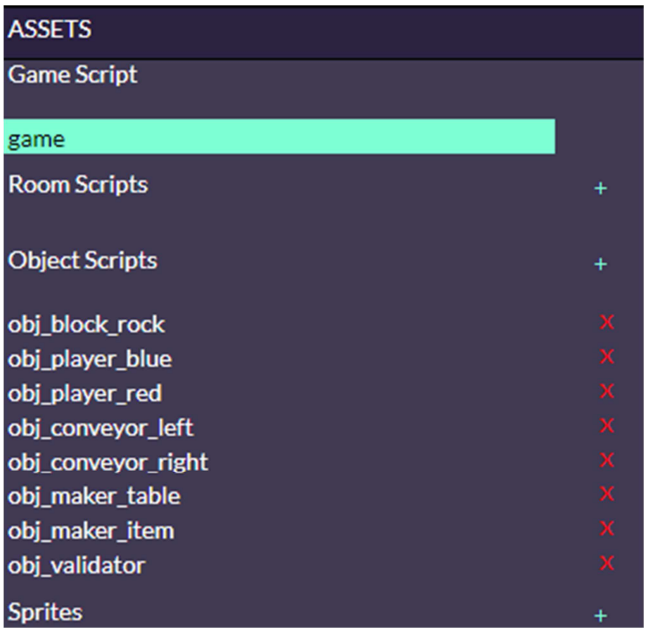
Create Objects

Start of App Guide

First we need to create the objects we're going to use in our game. If you forget how to add objects and/or assets into PixelPAD you can always refer to the introductory lessons in an earlier chapter.

Add the following objects into the game:

- obj_block_rock
- obj_player_blue
- obj_player_red
- obj_conveyor_left
- obj_conveyor_right
- obj_maker_table
- obj_maker_item
- obj_validator



We will be referring to these assets throughout the tutorial.

Tiling using For Loops

Game Script – Start tab

```
# Enter the start code for your game here.

for xx in range(-10, 10):
  for yy in range(-10,10):
    ground_tile = object_new("obj_block_rock")
    ground_tile.sprite = sprite_new("spr_block_rock")
    ground_tile.x = xx * 63
    ground_tile.y = yy * 50
    ground_tile.z = -yy
```

If you're new to PixelPAD and Python, this may be a fairly difficult concept to understand. It is highly recommended to move on and return to this once you have a better understanding of for loops, object creation, sprites, and x, y, z dot notation.

Here's an explanation of what's happening in the code if you're up for the challenge!

To understand what's happening here, we need to take a look at each loop individually; if we take a simplified version of it, let's see what happens.

```
for xx in range(-10,10):
   print(xx)
```

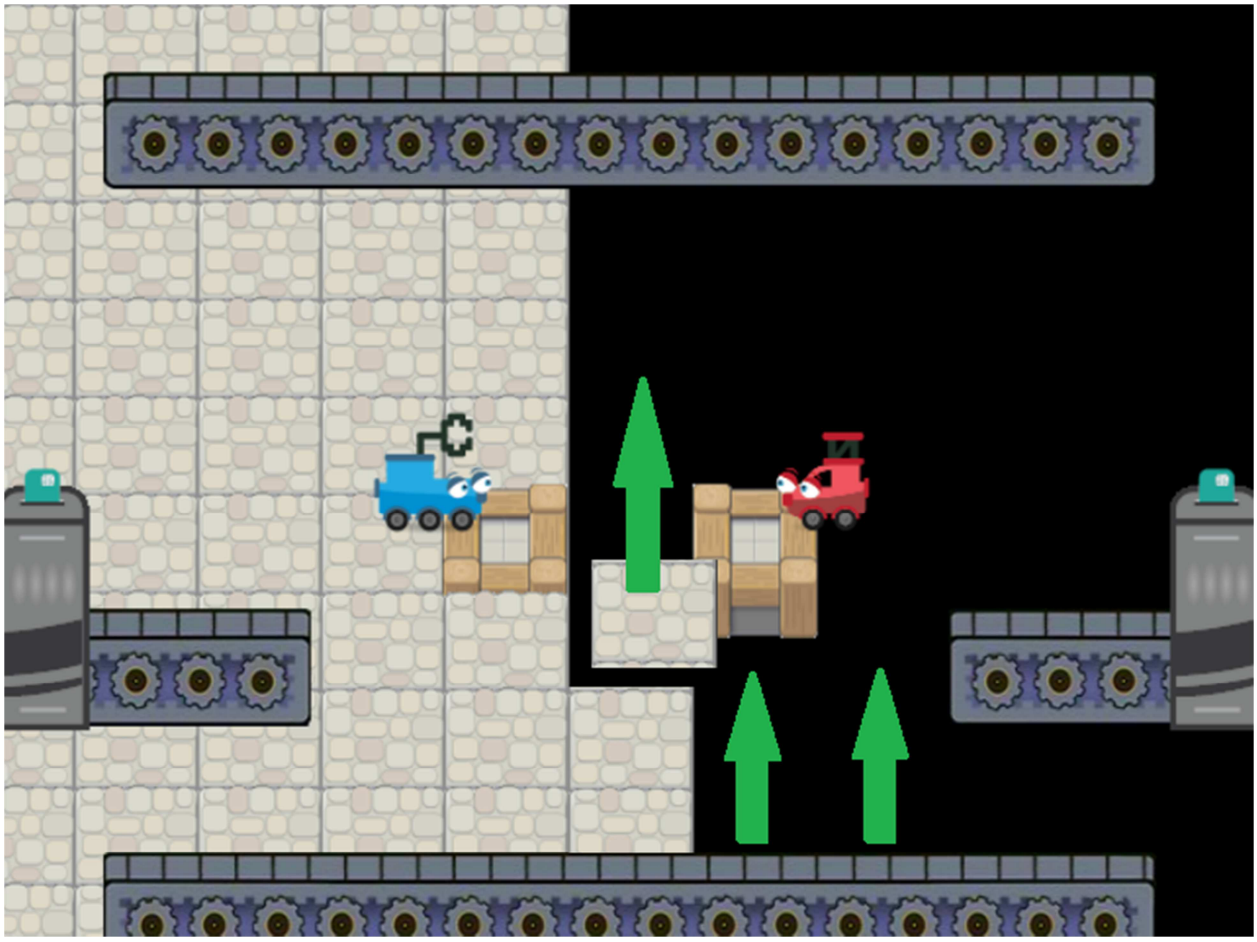
This would print -10,-9,-8,-7,-6… 9.

Now the other for loop is almost identical except it uses yy instead of xx as its variable.

The tricky part is understanding what happens when you put one for-loop within another. By having one for loop within another, the loop on the outside does not move onto the next number until the loop on the inside has finished.

Tiling using For Loops

The figure below shows how the tiles are placed. The tiles are placed bottom up, from yy =-10 to yy = 9, then moving over to the right by 1, and starting over again at yy = -10



We multiply by xx and yy by 63 and 50 respectively because that's the length and width of the tile we're placing down.

Adding Maker Tables

Game Script – Start Tab

```
maker_table = object_new("obj_maker_table")
maker_table.sprite = sprite_new("spr_maker_table")
maker_table.x = -64
maker_table.y = -45
maker_table.z = 1

maker_table = object_new("obj_maker_table")
maker_table.sprite = sprite_new("spr_maker_table")
maker_table.x = 64
maker_table.y = -45
maker_table.z = 1
```

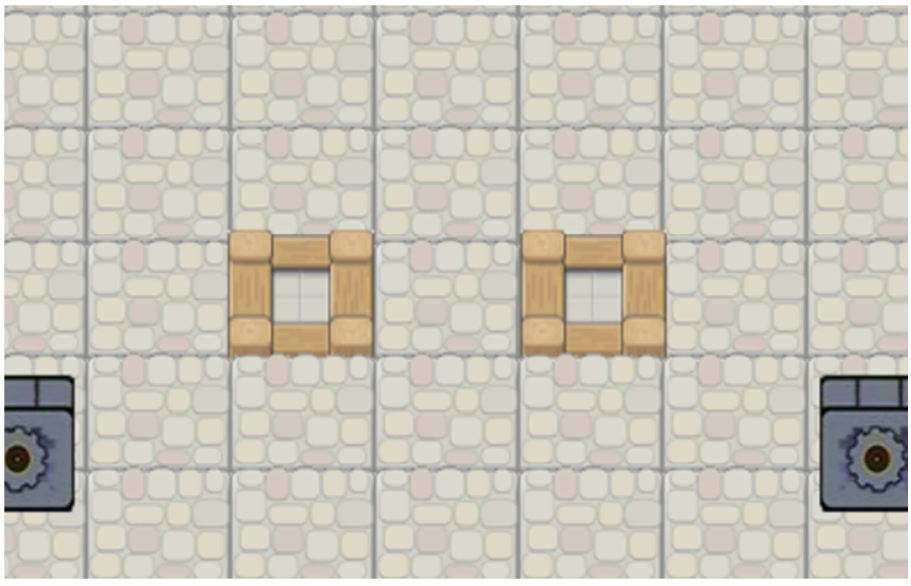
This creates the maker tables in the center of the screen. This is where we allow the players to craft items!

The first line creates a new object called "obj_maker_table" and labels it maker_table.

Then we give maker_table the sprite spr_maker_table

Lastly we move the maker table to the coordinates (-64, -45, 1) and (64, -45, 1) respectively.

The code for the first maker table and the second maker table is almost exactly identical, can you spot the difference?



Adding Player 1

Game Script - start tab

```
p1 = object_new("obj_player_red")
p1.sprite = sprite_new("spr_player_red")
p1.z = 1000
p1.sprite_height = 0.5
p1.sprite_width = 0.5
p1.x = 100
p1.name = "p1"
p1.score = 0
```

This does very similar things to what we did with the maker table except we are adding a few more variables here.

We changed the z value for the player to be 1000, because we always want the player to be showing above all other assets.

We also gave the player a name called "p1". This can be used later when we need to print the player's name.

We also added a new variable called "score" that we will use to keep track of the player's score

Now it's your turn! Try to add player 2 to your game in the "Game Script"! Use "p2" instead of "p1" as your reference to the object. I would also use a different colour for p2, but that's up to you!

Game Script - start tab

```
p2 = object_new("obj_player_blue")
p2.sprite = sprite_new("spr_player_blue")
p2.z = 1000
p2.x = -100
p2.sprite_height = 0.5
p2.sprite_width = 0.5
p2.name = "p2"
p2.score = 0
```

Conveyors

Game Script - Start Tab

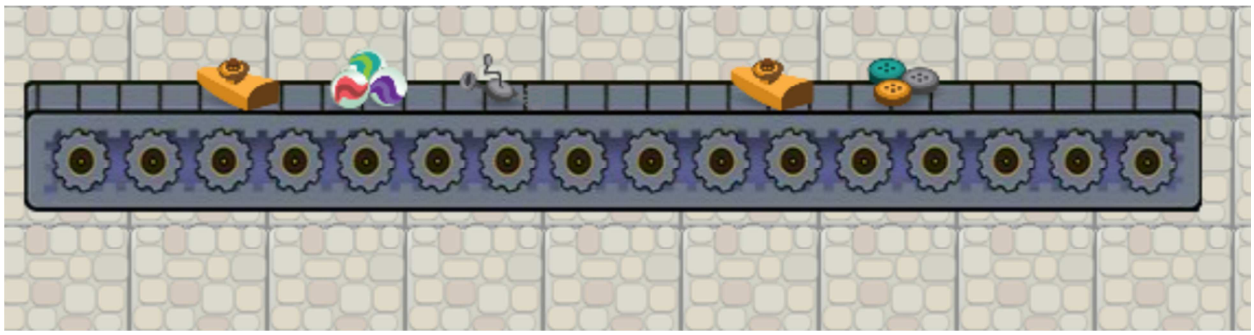
```
conveyor_left = object_new("obj_conveyor_left")
conveyor_left.y = 175
conveyor_left.z = 900
conveyor_left.sprite_width = 0.89
conveyor_left_sheet = sprite_new("spr_conveyor_left", 1, 4)
conveyor_left_anim = animation_new(conveyor_left_sheet, 15, 0,3)
animation_set(conveyor_left, conveyor_left_anim)
```

We need to create two conveyors that will be moving our objects to the left and right! This is code creates the left conveyor that brings our maker items in.

Note: It's called conveyor_left because it moves our items left

The only part of this code that should be new to you is the animation, so let's talk about the 3 functions that make animation work, sprite_new, animation_new and animation_set.

➔ Continued on the next page ➔



Conveyor Animations

Game Script - Start Tab

```
conveyor_right = object_new("obj_conveyor_right")
conveyor_right.y = -225
conveyor_right.z = 900
conveyor_right.sprite_width = 0.89
conveyor_right_sheet = sprite_new("spr_conveyor_right", 1, 4)
conveyor_right_anim = animation_new(conveyor_right_sheet, 15, 0,3)
animation_set(conveyor_right, conveyor_right_anim)
```

This code creates the right conveyor that brings our maker items in.

sprite_new needs 3 arguments, the location of the sprite and the number of columns and rows this sprite has.

Considering the sprite sheet for the conveyors looks like this:

The number of rows is 1 and number of columns is 4 - that is, there is one row of images and 4 images within that row.

animation_new takes 4 arguments, the sprite sheet, speed of animation, frame start and frame end.

Here we named our sprite_sheet "conveyor_right_sheet" and "conveyor_left_sheet" and we set animation speed to 15 with the starting frame at 0 and the ending frame at 3.

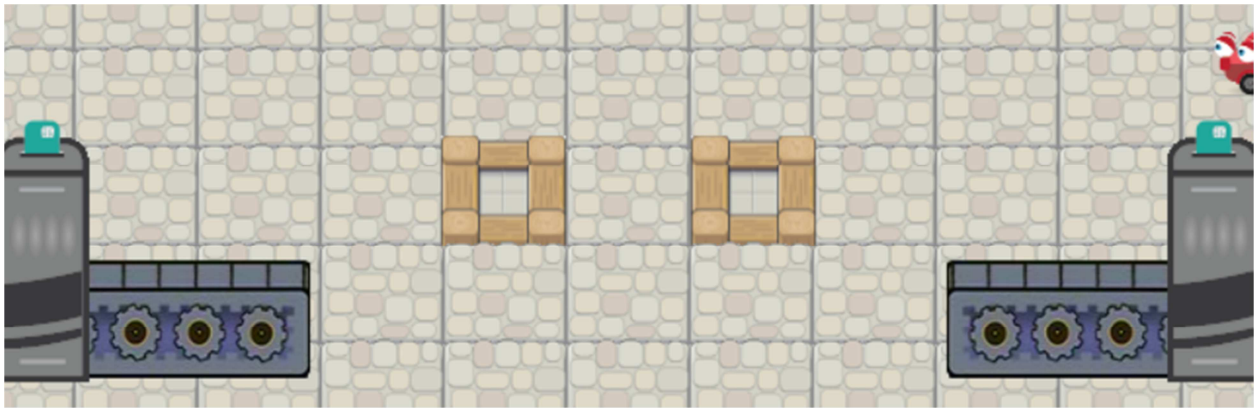
Lastly animation_set sets the animation. Animation set takes 2 arguments, the object you want to set the animation for, and the animation you want to set it to.

Left and Right Conveyors

Now it's your turn! We need 2 more conveyors to the left and right of the players; these will be the conveyors that maker items are placed to be carried out and scanned for points!

We also need to include two "validators" that will scan our objects at the end of our two conveyors. Use the variables "validator_left" and "validator_right" to store your validator objects.

Can you figure out how to code these items on your own?



Game Script – Start Tab

```
conveyor_left = object_new("obj_conveyor_left")
conveyor_left.x = -432
conveyor_left.y = -100
conveyor_left.z = 900
conveyor_left.sprite_width = 0.89
conveyor_left_sheet = sprite_new("spr_conveyor_left", 1, 4)
conveyor_left_anim = animation_new(conveyor_left_sheet, 30, 0,3)
animation_set(conveyor_left, conveyor_left_anim)

conveyor_right = object_new("obj_conveyor_right")
conveyor_right.x = 432
conveyor_right.y = -100
conveyor_right.z = 900
conveyor_right.sprite_width = 0.89
conveyor_right_sheet = sprite_new("spr_conveyor_right", 1, 4)
conveyor_right_anim = animation_new(conveyor_right_sheet, 30, 0,3)
animation_set(conveyor_right, conveyor_right_anim)
```

TIP: Make the animation of the conveyors different to add a little variety to your game!

Validators

Game Script – Start Tab

```
maker_item_timer = 30

validator_left = object_new("obj_validator")
validator_left.x = -300
validator_left.y = -65
validator_left.z = 2000
validator_left.sprite = sprite_new("spr_validator")
validator_left.player = p1

validator_right = object_new("obj_validator")
validator_right.x = 300
validator_right.y = -65
validator_right.z = 2000
validator_right.sprite = sprite_new("spr_validator")
validator_right.player = p2
```

Creating Items at Random

Game Script – Loop Tab

```
# Enter the loop code for your game here.
import random

maker_item_timer -=1
if maker_item_timer < 0:
  maker_item = object_new("obj_maker_item")
  maker_item.sprite_height = 0.33
  maker_item.sprite_width = 0.33
  maker_item.z = 1000
  random_conveyor = random.randint(0,1)
  if random_conveyor == 0:
    maker_item.x = 250
    maker_item.y = 210
  else:
    maker_item.x = -250
    maker_item.y = -180
```

Here we need to import "random" so we can use the function random.randint() to give us a random number. We'll use this random number to create random objects and at random conveyors!

Once our maker_item_timer counts down, we create a new obj_maker_item and depending on a random number 0 or 1, we place it either on the top conveyor or the bottom conveyor.

Creating Items at Random

Game Script – Loop Tab

```
if maker_item_timer < 0:
  random_sprite = random.randint(0,6)

  if random_sprite == 0:
    maker_item.sprite = sprite_new("spr_beater")
    maker_item.name = "beater"
  elif random_sprite == 1:
    maker_item.sprite = sprite_new("spr_buttons")
    maker_item.name = "buttons"
  elif random_sprite == 2:
    maker_item.sprite = sprite_new("spr_hanger")
    maker_item.name = "hanger"
  elif random_sprite == 3:
    maker_item.sprite = sprite_new("spr_yarn")
    maker_item.name = "yarn"
  elif random_sprite == 4:
    maker_item.sprite = sprite_new("spr_marbles")
    maker_item.name = "marbles"
  elif random_sprite == 5:
    maker_item.sprite = sprite_new("spr_wires")
    maker_item.name = "wires"
  elif random_sprite == 6:
    maker_item.sprite = sprite_new("spr_teeth")
    maker_item.name = "teeth"
  maker_item_timer = 60
```

Here we use random.randint to decide on which items we want to make!

We choose a random number between 0 and 6 and depending on what number comes up; we change the sprite of the maker item and give it a "name".

We'll use the name later as an easy way to identify which maker item we're working with!

Here you get creative; you can add your own maker items into the game by finding your own images online, or even changing how often an item spawns!

Moving Player 1

obj_player_blue - Loop Tab

```
# Enter the loop code for obj_player_blue here.
self.spd = 5

if key_is_pressed("a"):
  self.x -= self.spd

if key_is_pressed("d"):
  self.x += self.spd

if key_is_pressed("w"):
  self.y += self.spd

if key_is_pressed("s"):
  self.y -= self.spd

if key_was_pressed("space"):
  if held_item:
    held_item = 0
  else:
    held_item = collision_check(self, "obj_maker_item")

if held_item:
  held_item.x = self.x
  held_item.y = self.y
  held_item.direction = "none"
```

key_is_pressed("") checks if a key is currently being pressed. This function checks every frame in the loop.

If you only want it to check once, then use key_was_pressed("") i.e. in the case of "if key_was_pressed("space")" we are only checking if the user presses space once.

If the player presses space, we check if they are holding an item. If the player is holding an item, drop it or in code... set held_item to 0.

If he's NOT holding an item, then pick up the item he's currently colliding with.

Lastly, we check if the player is holding an item. If he is, we want to change the position of the item to wherever the player is.

Moving Player 2

Now it's your turn! Try to add the code for the second player to your game. Instead of using "wasd" to move, use arrow keys, and instead of using "space" to pick up the item, use "enter".

obj_player_red - Loop Tab

```
# Enter the loop code for obj_player_red here.

self.spd = 3

if key_is_pressed("left"):
  self.x -= self.spd

if key_is_pressed("right"):
  self.x += self.spd

if key_is_pressed("down"):
  self.y -= self.spd

if key_is_pressed("up"):
  self.y += self.spd

if key_was_pressed("enter"):
  if held_item:
    held_item = 0
  else:
    held_item = collision_check(self, "obj_maker_item")

if held_item:
  held_item.x = self.x
  held_item.y = self.y
  held_item.direction = "none"
```

Defining New Functions

obj_maker_table – Start Tab

```
# Enter the start code for obj_maker_table here.

def item_on_table(item_name):
  for item in collision_check_all(self, "obj_maker_item"):
    if item.name == item_name:
      return item
```

A function is a block of organized, reusable code that is used to perform a single, related action.

You may not believe it, but we've been using functions all along! Functions like: object_new(), sprite_new(), key_is_pressed() and key_was_pressed(), these were all pre-built functions we used to make our lives easier.

But now we get to define our own function!

The code above defines a new function named (item_on_table) which checks if there's a specific item on a table and returns the item.

The function works by checking all items the maker table is currently touching and if its name "item.name" is the same as the specific item we're looking for "item_name" then returns the item. If you don't quite understand how this works yet, don't worry! Just keep practicing!

Here's an example of a new function we can define that may be easier to understand that creates a new objects and places it in the room for us.

```
def object_new_ext(object, sprite, xx, yy):
  new_obj = object_new("object")
  new_obj.sprite = sprite_new("sprite")
  new_obj.x = xx
  new_obj.y = yy
  return new_obj
```

After defining this new function, we can place objects into our room with one line of code instead 4 like so:

```
maker_table = object_new_ext("obj_maker_table", "spr_maker_table", 100, 100)
```

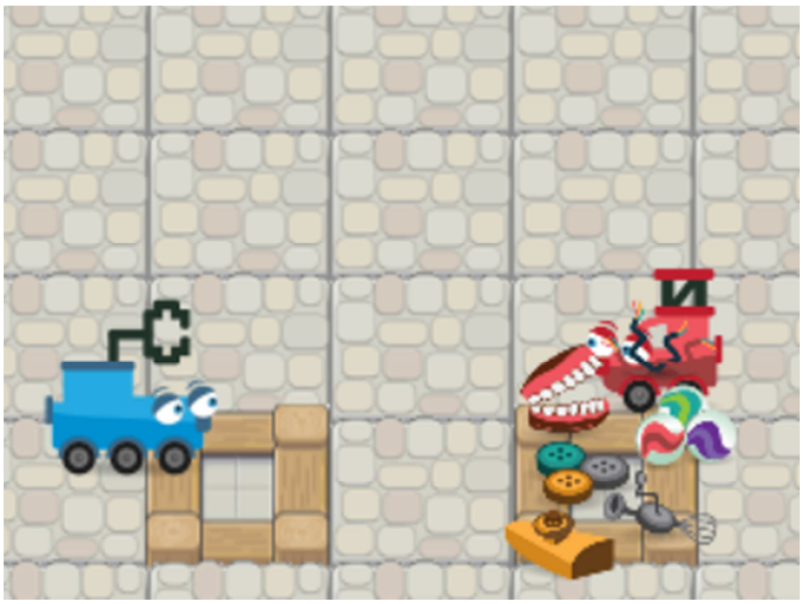
New Function: Combine Items

obj_maker_table – Start Tab

```
def combine_items(item_1, item_2, item_3, item_3_sprite):
  if item_on_table(item_1) and item_on_table(item_2):
    new_item = object_new("obj_maker_item")
    new_item.sprite = sprite_new(item_3_sprite)
    new_item.name = item_3
    new_item.x = self.x
    new_item.y = self.y
    new_item.z = 1000
    destroy(item_on_table(item_1))
    destroy(item_on_table(item_2))
```

This new function we're defining uses our previous function "item_on_table" to detect if 2 very specific items are on the table. If they are, combine them together to create a 3rd item.

"Combining" in this case actually means creating a new item, while destroying the 2 items on the table we used to combine.



Defining New Combinations

obj_maker_table – Loop Tab

```
# Enter the loop code for obj_maker_table here.

if key_was_pressed("space") or key_was_pressed("enter"):
  combine_items("teeth", "wires", "robot", "spr_robot")
  combine_items("yarn", "kazoo", "rocket", "spr_rocket")
  combine_items("buttons", "yarn", "wires", "spr_wires")
```

By defining the previous function "combine_items" we can now use it to combine items together!

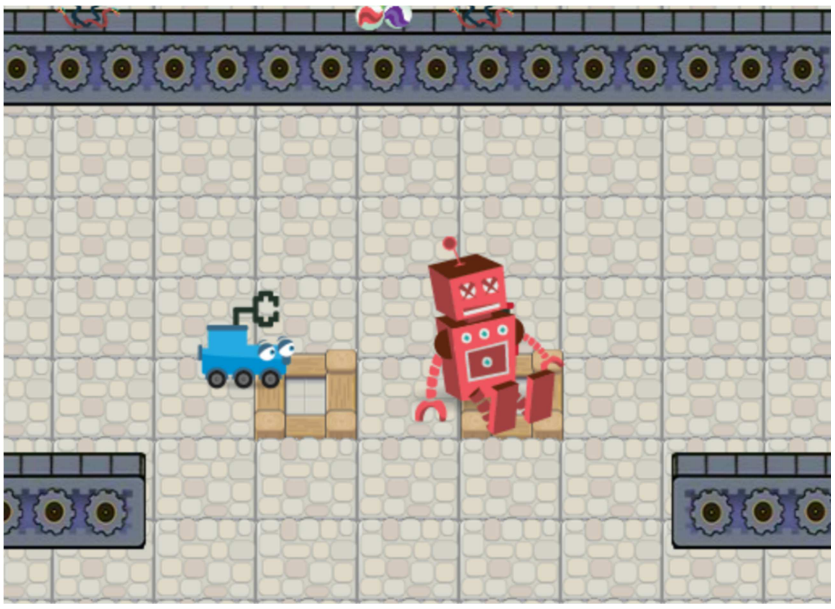
We check if there are items we can combine only after we pressed the space or enter buttons.

Then define which items we need to combine. If you look at our previous function we created, it takes 4 arguments:

```
def combine_items(item_1, item_2, item_3, item_3_sprite):
```

item_1 and item_2 combine creating item_3. Item_3_sprite is the sprite we use for item 3

Here you get to define as many combinations as you'd like! See if you can define a few more yourself.



Moving Items on Conveyor

obj_maker_item – Loop Tab

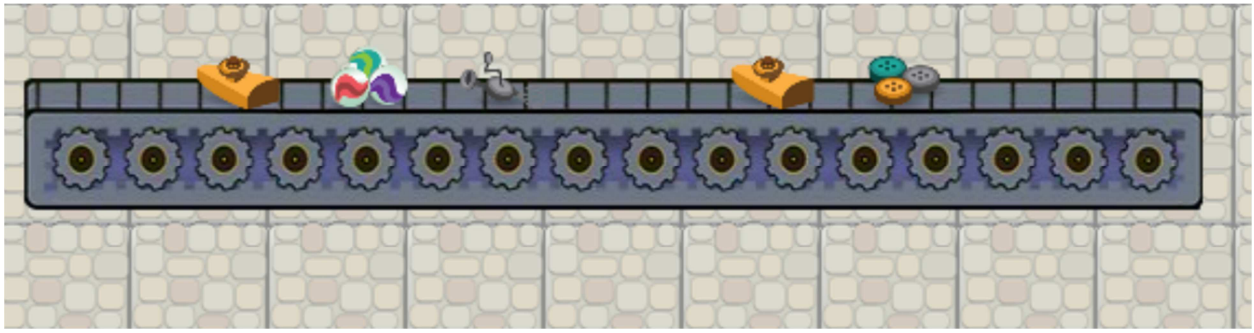
```
# Enter the loop code for obj_maker_item here.

if collision_check(self, "obj_conveyor_left"):
  self.x -= 1
if collision_check(self, "obj_conveyor_right"):
  self.x +=1

if self.x > 1000 or self.x < -1000:
  destroy(self)
```

If you try placing items on the conveyor belt right now, you'll notice you won't get much movement.

These statements will check if our item is currently on a conveyor belt. If it is move them in the right direction!



Score

obj_validator – Loop Tab

```
# Enter the loop code for obj_validator here.

given_item = collision_check(self, "obj_maker_item")

if given_item:
  if not given_item.counted:
    if given_item.name == "robot":
      player.score += 5
    elif given_item.name == "marbles":
      player.score += 1
    elif given_item.name == "rocket":
      player.score += 5
    given_item.counted = True
    print(player.name + " score: " + player.score)
```

Scoring! Now we're going to score our game. Remember those "validators" or scanners we created right at the start? We're going to use those to add points for our players.

The tricky part here is the line:

`If not given_item.counted:`

This checks for the variable called "counted" whenever there's a collision with the validator. We check **`if not counted`** to prevent the validator adding points for every frame the item touches the validator.

Challenge Questions

- Can you figure out how to add 3 additional maker items to your game?
- Can you figure out how to add 3 additional combinatory items to your game?
- Can you figure out how to add a time limit to your game?
- Can you figure out how to add a WIN screen for your game?
- In your own words, explain what a function is and how you would use it in this game to help you?
- Consider the following function

```
def object_new_ext(object, sprite, xx, yy):
  new_obj = object_new("object")
  new_obj.sprite = sprite_new("sprite")
  new_obj.x = xx
  new_obj.y = yy
  return new_obj
```

  How would you modify this function to set the objects "z" value?

  How would you modify this function to set the objects sprite_width and sprite height?