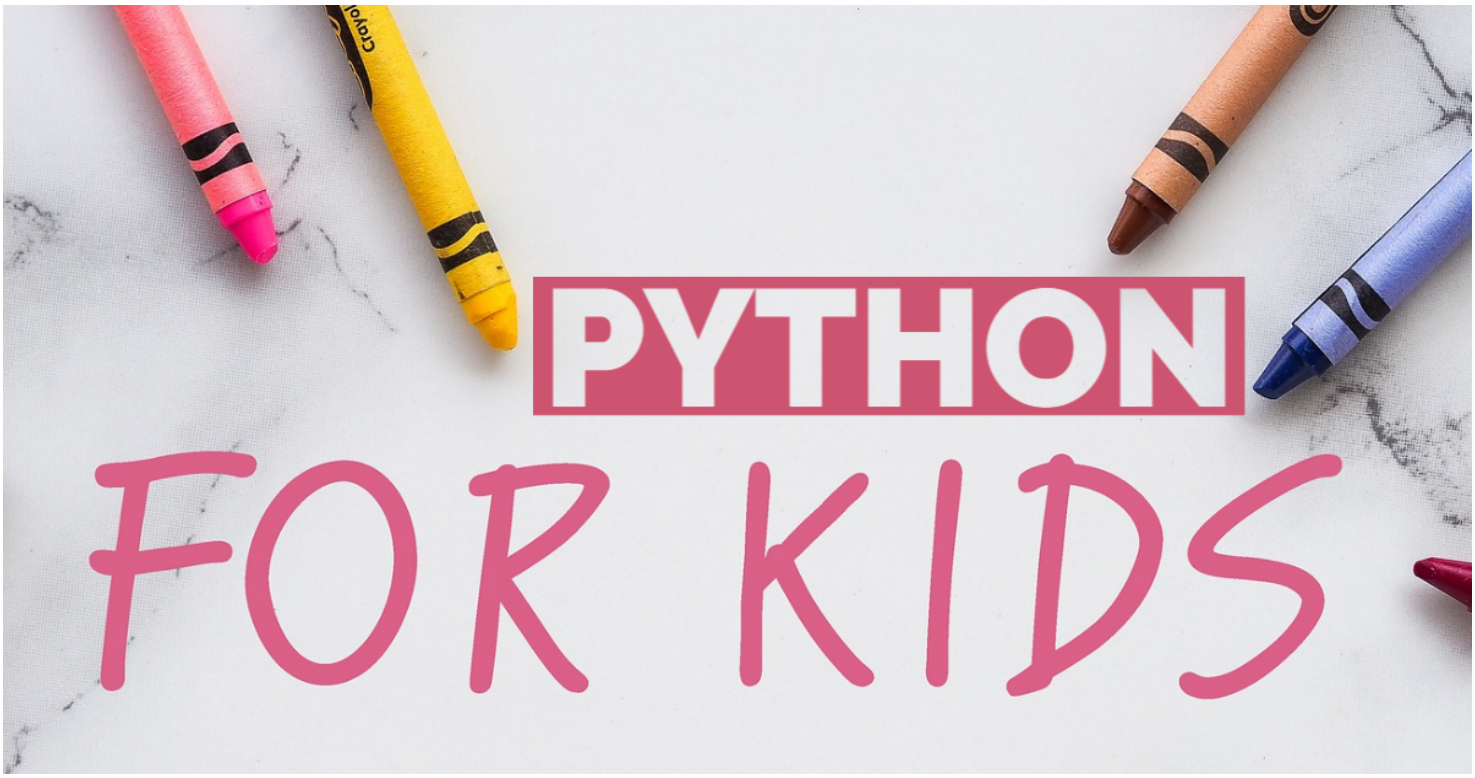


MICROPYTHON FIRST EDITION – 0.1 release

Kevin Thomas & Katie Henry

Forward

Over the next decade we will personally witness the full Cyber and Automation revolution in a way even twenty years ago would never have been thought possible.

The future careers will be focused primarily on Computer Engineering. It is fundamental that YOU as a parent have a full appreciation and understanding of this reality and the fact that YOUR child MUST start early in their Engineering development if they are going to thrive in the coming years.

Python For Kids is a bite-sized, step-by-step FREE book and video series designed for EVERYONE. As a parent YOU can on-demand work through these projects together and give them a gift that you would otherwise never be able to provide which is providing them the foundational knowledge to get started in the world of microcontrollers.

I want to thank Katie Henry who is the Head Of Partnerships For North America at the Microbit Educational Foundation for her brilliant contribution and video series for this book.

Katie is also a Former Classroom Teacher and has the experience and vision necessary to help teach and inspire the next generation of Engineers!

If you have questions you can also contact us on our subreddit **r/micropython** as well.

Table Of Contents

Chapter 1: Goals

Welcome to a brand-new series for kids who want to learn the FUN way to program in Python.

This tutorial will be a step-by-step instruction series where parents can learn to code with their kids and non-technical educators with their students and enrich their lives and prepare them for the next generation of work when they mature into adults.

Our goal is to teach kids to have FUN building simple projects and getting familiar with Python in a way that opens their mind toward technical creativity in ways they would never have experienced before! If you are parent this is a great opportunity for you to help set your child into the drivers seat as over the next decade we will continue to transition to a world which is significantly more automated to which future generations (YOUR KIDS) will need to have an upper hand when understanding and interacting with technology in ways you or I never had to imagine if they are to sustain in the coming years.

There will be three things you need to partake in this series. The first is an coding tool we will use to write our simple programs which is FREE provided by the micro:bit Educational Foundation to which we will access via the web.

STEP 1 - Purchase micro:bit V2

Please make sure you select the micro:bit v2 microcontroller from the link below and make sure to check if the USB cable is included in the purchase as that may be sold separately. You can select from any of the official micro:bit retailers.

https://microbit.org/buy

These are the only tools you will need to embark on this incredible journey to get your child prepared for the next IoT generation and solidify their role as a leader in technology!

In our next lesson we will work step-by-step on working with the FREE online micro:bit MicroPython Web Editor to begin programming!

Chapter 2: “Hello World”

Today we will set up our development environment so we can write simple and FUN programs with our new micro:bit so let's get started!

We will then create our first program!

STEP 1: Navigate To Web Editor

https://python.microbit.org/v/beta

```
# Add your Python code here. E.g.
from microbit import *


while True:
    display.scroll('Hello, World!')
    display.show(Image.HEART)
    sleep(2000)
```

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

STEP 4: Click "BBC micro:bit CMSIS-DAP" & CONNECT

STEP 5: Rename Script Name To helloworld_program

STEP 6: Click Flash

STEP 7: WATCH THE micro:bit V2 SCROLL 'Hello, World!' & SHOW A HEART!

Congrats! You did GREAT! You made your first program on the micro:bit V2 and you did a FANTASTIC job!

In our next lesson we will have FUN With Images!

Chapter 3: FUN With Images

Today we are going to make some FUN little images for our little friend to make and interact with us.

If your micro:bit is not connected to the micro:bit MicroPython Web Editor, please follow the steps in Lesson 2, specifically **STEP 2** and **STEP 3**:

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

If you are connected you are all ready to start!

STEP 1: Open micro:bit MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Type Code & Name It images_program

```
from microbit import *


display.show(Image.HAPPY)
```

STEP 3: Click Save

STEP 4: Click Download Python Script

STEP 5: Click Flash

STEP 6: SEE THE FUN AS OUR NEW FRIEND MAKES A SMILE AT US!

(Look at the front of the badge to see the image.)

STEP 7: Let's Create Our OWN Image!

Let's create our very OWN shape! Each LED pixel on the physical display can be set to one of ten values. If a pixel is set to 0 (zero) then it’s off. It literally has zero brightness. However, if it is set to 9 then it is at its brightest level. The values 1 to 8 represent the brightness levels between off (0) and full on (9).

Let's create a little house. We can start by overwriting our image_program file and naming it **my_image_program** and type the following into the editor.

```
from microbit import *


house = Image("00900:"
              "09090:"
              "90009:"
              "05550:"
              "05950")


display.show(house)
```

STEP 8: Click Save

STEP 9: Click Download Python Script

STEP 10: Click Flash

STEP 11: SEE THE FUN AS WE JUST MADE A LITTLE HOUSE!

(Look at the front of the micro:bit to see the image.)

It's your turn! Ask our little friend to draw some of his other favorite images for you! You can try something like this on line 3 by replacing line 3 in our code with one of the below statements to which you will SAVE, DOWNLOAD and FLASH as described above.

You can name each new program anything you like! For example you could do the following for the heart image and after you SAVE, DOWNLOAD and FLASH you can start the process over and try the next one till you get through all of the examples below.

Here’s a list of the built-in images:

```
display.show(Image.HEART)
display.show(Image.HEART_SMALL)
display.show(Image.SMILE)
display.show(Image.SAD)
display.show(Image.CONFUSED)
display.show(Image.ANGRY)
display.show(Image.ASLEEP)
display.show(Image.SURPRISED)
display.show(Image.SILLY)
display.show(Image.FABULOUS)
```

```
display.show(Image.MEH)
display.show(Image.YES)
display.show(Image.NO)
display.show(Image.TRIANGLE)
display.show(Image.TRIANGLE_LEFT)
display.show(Image.CHESSBOARD)
display.show(Image.DIAMOND)
display.show(Image.DIAMOND_SMALL)
display.show(Image.SQUARE)
display.show(Image.SQUARE_SMALL)
display.show(Image.RABBIT)
display.show(Image.COW)
display.show(Image.MUSIC_CROTCHET)
display.show(Image.MUSIC_QUAVER)
display.show(Image.MUSIC_QUAVERS)
display.show(Image.PITCHFORK)
display.show(Image.XMAS)
display.show(Image.PACMAN)
display.show(Image.TARGET)
display.show(Image.TSHIRT)
display.show(Image.ROLLERSKATE)
display.show(Image.DUCK)
display.show(Image.HOUSE)
display.show(Image.TORTOISE)
display.show(Image.BUTTERFLY)
display.show(Image.STICKFIGURE)
display.show(Image.GHOST)
display.show(Image.SWORD)
display.show(Image.GIRAFFE)
display.show(Image.SKULL)
display.show(Image.UMBRELLA)
display.show(Image.SNAKE)
```

Feel free to share your code in the comments below! I would LOVE to see what YOU create!

In our next lesson we will have some FUN with numbers!

Chapter 4: FUN With Numbers

Today we are going to have FUN with our little friend and work with numbers and make a simple adding and subtracting calculator!

We will use what we call a variable to hold a number for as long as you might want. Think of a variable as a little tiny box that you can put a number in and do something with and then take it out and put another number in to replace it.

In our case we are simply going to create a variable and call it counter to which we will set to 0 and use the A button to add 1 to and the B button to subtract 1 from.

Let's teach our little friend to add and subtract numbers by clicking its buttons. When you press the A button it will add a number and scroll it on his little screen and when you press the B button it will subtract a number and scroll it on his little screen.

Let's look at a picture of our little friend and see where the A and B buttons are. This should look familiar as this is a picture from our "Hello World" lesson.

Let's have some FUN and code up our little friend to be a little calculator!

If your micro:bit is not connected to the micro:bit MicroPython Web Editor, please follow the steps in Lesson 2, specifically **STEP 2** and **STEP 3**:

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

If you are connected you are all ready to start!

STEP 1: Open micro:bit MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Type Code & Name It numbers_program

```
from microbit import *


counter = 0


while True:
    if button_a.was_pressed():
        counter = counter + 1
        display.scroll(str(counter))
    if button_b.was_pressed():
        counter = counter - 1
        display.scroll(str(counter))
```

STEP 3: Click Save

STEP 4: Click Download Python Script

STEP 5: Click Flash

STEP 6: SEE THE FUN AS OUR NEW FRIEND WILL ADD AND SUBTRACT NUMBERS WHEN WE PRESS ITS BUTTONS!

(Look at the front of the badge to see the image.)

Now that you have created a little calculator how about you try some FUN new things on your own! Go back to the code and on line 7 try changing it to something like the below statement to which you will SAVE, DOWNLOAD and FLASH as described above.

```
counter = counter + 10
```

You can also change the number or variable on line 10 by changing it to something like the below statement to which you will SAVE, DOWNLOAD and FLASH as described above.

```
counter = counter - 10
```

Feel free to share your code in the comments below! I would LOVE to see what YOU create!

In our next lesson we will have some FUN with words!

Chapter 5: FUN With Words

Today we are going to have FUN with our little friend and work with words and have him type our name and say hi to us!

We will use a word variable which we call a string to customize our little friends ability to say hello to us.

Instead of typing my name in **STEP 2** on line 3, as you can see below, type in your own name instead.

If your micro:bit is not connected to the micro:bit MicroPython Web Editor, please follow the steps in Lesson 2, specifically **STEP 2** and **STEP 3**:

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

If you are connected you are all ready to start!

STEP 1: Open micro:bit MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Type Code & Name It words_program

```
from microbit import *


name = 'Kevin'


while True:
    display.scroll('Hi ' + name + '!')
```

STEP 3: Click Save

STEP 4: Click Download Python Script

STEP 5: Click Flash

STEP 6: SEE THE FUN AS OUR NEW FRIEND SAYS HI TO US!

(Look at the front of the badge to see the image.)

Now that you taught our little friend to say hi to us how about you try some other words and have him say different things to us. You can try changing the below statement to something like this on line 6 to which you will SAVE, DOWNLOAD and FLASH as described above.

```
display.scroll(name + ', thank you for learning Python with me!  I cannot wait to see all the fun things we make!')
```

Feel free to share your code in the comments below! I would LOVE to see what YOU create!

In our next lesson we will have some FUN with word lists!

Chapter 6: FUN With Word Lists

Today we are going to have FUN with our little friend and work with word lists and have him tell us his favorite foods!

A word list is a collection of items that are all within one variable. This is very handy when you want to do more advanced programming in the future when dealing with Python lists which are nothing more than arrays.

If your micro:bit is not connected to the micro:bit MicroPython Web Editor, please follow the steps in Lesson 2, specifically **STEP 2** and **STEP 3**:

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

If you are connected you are all ready to start!

STEP 1: Open micro:bit MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Type Code & Name It word_lists_program

```
from microbit import *


favorite_foods = ['pizza', 'ice cream', 'cookies']


while True:
    display.scroll(
                   'I love to eat ' +
                   favorite_foods[0] + ', ' +
                   favorite_foods[1] + ', ' +
                   'and ' +
                   favorite_foods[2] + '!'
                  )
```

STEP 3: Click Save

STEP 4: Click Download Python Script

STEP 5: Click Flash

STEP 6: SEE THE FUN AS OUR NEW FRIEND TELLS US ALL HIS FAVORITE FOODS!

(Look at the front of the badge to see the image.)

Now that you taught our little friend to tell us all his favorite foods how about you type in your three favorite foods. You can try something like this on line 3 however type in your three favorite foods instead to which you will SAVE, DOWNLOAD and FLASH as described above.

```
favorite_foods = ['candy', 'chips', 'cake']
```

Feel free to share your code in the comments below! I would LOVE to see what YOU create!

In our next lesson we will have some FUN with sounds!

Chapter 7: FUN With Music

IS EVERYONE EXCITED? IS EVERYONE READY TO GET THEIR JAM ON?

Today we are going to have FUN with our little friend and have him jam out some tunes for us as the party starts right now!

If your micro:bit is not connected to the micro:bit MicroPython Web Editor, please follow the steps in Lesson 2, specifically **STEP 2** and **STEP 3**:

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

If you are connected you are all ready to start!

STEP 1: Open micro:bit MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Type Code & Name It music_program

```
from music import *


play(NYAN)
```

STEP 3: Click Save

STEP 4: Click Download Python Script

STEP 5: Click Flash

STEP 6: SEE THE FUN AS OUR NEW FRIEND IS JAMMIN OUT HIS FAVORITE TUNE!

It's your turn! Ask our little friend to play some of his other favorite songs for you! You can try something like this on line 3 by replacing line 3 in our code with one of the below statements to which you will SAVE, DOWNLOAD and FLASH as described above.

Here’s a list of the built-in songs:

play(DADADADUM)
play(ENTERTAINER)

```
play(PRELUDE)
play(ODE)
play(RINGTONE)
play(FUNK)
play(BLUES)
play(BIRTHDAY)
play(WEDDING)
play(FUNERAL)
play(PUNCHLINE)
play(PYTHON)
play(BADDY)
play(CHASE)
play(BA_DING)
play(WAWAWAWAA)
play(JUMP_UP)
play(JUMP_DOWN)
play(POWER_UP)
play(POWER_DOWN)
```

Feel free to share your code in the comments below! I would LOVE to see what YOU create!

In our next lesson we will have some FUN with talking robots!

Chapter 8: FUN With Talking Robots

Today...

We...

Build...

An...

Interactive...

Robot...

Today we are going to have the MOST FUN YET by chatting to Mr. George who is a robot who lives inside our microcontroller and he CAN'T WAIT to talk to you!

If your micro:bit is not connected to the micro:bit MicroPython Web Editor, please follow the steps in Lesson 2, specifically **STEP 2** and **STEP 3**:

STEP 2: Connect Your micro:bit V2 Into Your Computer

STEP 3: Press The Connect Button In The micro:bit MicroPython Web Editor

If you are connected you are all ready to start!

STEP 1: Open micro:bit MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Type Code & Name It talking_robots_program

```
import gc

import speech


def talk(words):
    """Talk to your friend Mr. George

    Parameters
    ----------
    words : str
        The words to say to your friend Mr. George
    Returns
```

```
    -------
    None
    """
    gc.collect()
    words = words.lower()

    if words.find('how are you') != -1:
        speech.say('I am doing great!')
    elif words.find('what\'s up') != -1:
        speech.say('The sky.')
    elif words.find('morning') != -1:
        speech.say('I love to watch the sun rise in the morning!')
    elif words.find('afternoon') != -1:
        speech.say('I get hungry around lunch time.')
    elif words.find('evening') != -1:
        speech.say('I get sleepy in the evening.')
    elif words.find('night') != -1:
        speech.say('I get sleepy when it is night time.')
    elif words.find('tell me something') != -1:
        speech.say('I am a robot who loves to teach Piethon.')
    elif words.find('hello') != -1:
        speech.say('Hello to you!')
    elif words.find('hi') != -1:
        speech.say('Hi to you!')
    elif words.find('thank you') != -1:
        speech.say('It is my pleasure!')
    elif words.find('bye') != -1:
        speech.say('It was nice talking to you!')
    elif words.find('help') != -1:
        speech.say('I am always here to help!')
    elif words.find('what can you do') != -1:
        speech.say('I can teach Piethon programming.')
    elif words.find('name') != -1:
        speech.say('My name is Mr. George it is nice to meet you!')
    elif words.find('how old are you') != -1:
        speech.say('I was born in September of the year twenty twenty.')
    elif words.find('question') != -1:
        speech.say('I always try to answer questions.')
    elif words.find('joke') != -1:
        speech.say('What did the chicken cross the road?')
        speech.say('To get to the other side.')
    elif words.find('love') != -1:
        speech.say('I love pizza!')
    elif words.find('love you') != -1:
        speech.say('Thank you so kind of you!')
    elif words.find('love people') != -1:
        speech.say('I want to help people by teaching them Piethon!')
    elif words.find('hobby') != -1:
        speech.say('I like to teachin Piethon to people!')
    elif words.find('you live') != -1:
        speech.say('I live in side the little microcontroller here.')
    elif words.find('made you') != -1:
        speech.say('Kevin Thomas created me inspired by the great people at
MicroPiethon.')
    elif words.find('your job') != -1:
```

```
        speech.say('I teach Piethon.')
    elif words.find('you do') != -1:
        speech.say('I like to teach Piethon.')
    # ADD MORE CODE HERE
    else:
       pass
```

STEP 3: Click Save

STEP 4: Click Download Python Script

STEP 5: Click Flash

STEP 6: Click Open Serial

STEP 7: Type Into REPL The Following & Press Enter (Ask Mr. George A Question)

```
>>> import main
>>> main.talk('What is your name?')
```

STEP 8: HEAR MR. GEORGE TELL YOU HIS NAME!

CONGRATULATIONS! You have brought to LIFE your FIRST TALKING ROBOT WHICH YOU CAN TALK TO DIRECTLY!

It's your turn! Ask Mr. George some more questions! You can try something like this by re-typing into the REPL with one of the below statements and press enter.

Here’s a list of the built-in questions and things to say to Mr. George:

```
main.talk('How are you?')
main.talk('What's up?')
main.talk('Good morning!')
main.talk('Good afternoon!')
main.talk('Good evening!')
main.talk('Good night!')
main.talk('Hello Mr. George!')
main.talk('Hi Mr. George!')
main.talk('Thank you Mr. George!')
main.talk('Bye Mr. George!')
main.talk('Help me Mr. George!')
main.talk('What can you do?')
main.talk('How old are you?')
main.talk('Do you like questions?')
main.talk('Tell me a joke.')
```

```
main.talk('What do you love?')
main.talk('Do you have a hobby?')
main.talk('Where do you live?')
main.talk('Who made you?')
main.talk('What is your job?')
main.talk('What do you do?')
```

Feel free to share your questions in the comments below! I would LOVE to see what YOU create!

In our next lesson we will have some FUN learning about dozens of FREE additional STEP-BY-STEP lessons online and discuss how YOU can teach one of your friends what you just learned except YOU will be the teacher!

Chapter 9: Basic I/O

Now that we have the fundamentals in place let’s take it to the next level!

MicroPython is a complete re-implementation of CPython which is the Python you use on your typical computer. Both CPython and MicroPython are written in the C programming language.

MicroPython is designed for microcontrollers of which the micro:bit is. MicroPython is crafted to work directly with the device's architecture to create just about any application you can dream of.

We will also work with that we call the REPL which is integrated in the MicroPython Web Editor which gives us very special access to the microcontroller in a very special way. REPL stands for repeat, evaluate, print and loop such that we can do a great deal of debugging or code inspection there.

By the end of the lesson we will have completed the following.

```
* Written a 0001_hello_world_repl.py app which will output "Hello World!"
to the REPL or web terminal or console.

* Written a 0002_hello_world.py app which will display "Hello World!"
on our micro:bit display LED matrix.

* Written a 0003_hello_world_talk.py app to which our micro:bit will
display a talking image in addition to hearing the device speak the
words "Hello World!"

* Written a 0004_basic_io_repl.py app which will demonstrate the
ability for us to obtain keyboard input and dynamically populate
logic in the REPL based on the user submission.
```

STEP 1: Open The MicroPython Web Editor

https://python.microbit.org/v/beta

STEP 2: Plug-In micro:bit To Computer USB

STEP 3: Click Connect Button

STEP 4: Click BBC micro:bit CMSIS-DAP & Click Connect

STEP 5: Select All Code From Demo Program

STEP 6: Delete Demo Code

STEP 7: Rename File To 0001_hello_world_repl

STEP 8: Type Code

```
print('Hello World!')
```

STEP 9: Click Load/Save

STEP 10: Click Download Python Script [NOTE: Might Prompt Warning - IGNORE]

STEP 11: Click Flash

STEP 12: Click Open Serial

STEP 13: Review Output

```
Hello World!
MicroPython v1.13 on 2020-12-03; micro:bit v2.0.0-beta.2 with nRF52833
Type "help()" for more information.
>>>
```

We begin our understanding of MicroPython with the *print* function. The *print* function in MicroPython is a *built-in* function which literally prints strings or words into the REPL. In order for the *print* function to be executed we need to add a pair of parenthesis () after the function name.

The words or string that goes between the parenthesis are what is called a *function argument*. In our case, we are going to pass a string surrounded by a set of single or double quotes. In this course we will be using the single quote convention primarily as it is simply a design choice.

The contents of the *print* function is nothing more than *print('Hello World!')* which in this situation will print out simply the words Hello World! to the REPL. Whatever word or words we put inside the parenthesis will determine what gets printed to the REPL.

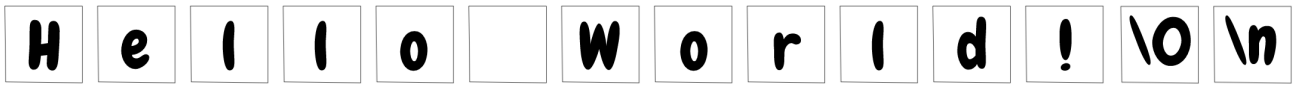
Let's dive into what a string is. A string is a string of characters or letters. Imagine if we had a bunch of little boxes on a table.

So we have our string, *Hello World!* which takes up 12 boxes.

Strangely if we count the boxes we see 14. Let's examine why.

Each box contains a letter or character which we refer to as an element in MicroPython. There is also what we call a null terminating character '\0' and a new line character '\n' that are two additional characters inside the print function. The good news is when the MicroPython team built MicroPython from C, they built-in what we refer to as *default arguments* inside the *print* function so you, the Developer, does not have to type them every time you want to print something to the REPL.

Now let's look at the boxes with all of the letters, spaces, null terminating character and new line character.



That is exactly what is going on inside your computer's memory under the hood.

When programming we all make typos or mistakes. If you leave out a quote you will get what is referred to a *SyntaxError* as you will notice the color scheme on that line will be slightly off. This is an indicator of a syntax error which is nothing more than a syntactical error when the MicroPython interpreter parses your line of code.

Let's look at an example:

```
print('Hello World!)
```

Notice we are missing the other quote mark after the exclamation point. Let's click Flash and Open Serial. Notice our little micro:bit is making a sad face and telling us something useful as well as what we see in the Serial REPL.

```
Traceback (most recent call last):
  File "main.py"
SyntaxError: invalid syntax
MicroPython v1.13 on 2020-12-03; micro:bit v2.0.0-beta.2 with nRF52833
Type "help()" for more information.
>>>
```

In addition, MicroPython will give you an error if you start a line of code that is not at the very beginning of the line as it will be

an indentation error as it will say it sees an unexpected indent error.

Let's look at an example:

```
print('Hello World!')
```

If you notice, the word print starts three spaces after it should.

Click Flash and Open Serial.

```
Traceback (most recent call last):
  File "main.py"
IndentationError: unexpected indent
```

We see our little micro:bit making another frown as well which is very helpful! We can see in the serial REPL an *IndentationError* as well.

Now that you have a handle on all of the steps to create, save, flash and REPL your code we will try some additional examples as well to help solidify these concepts.

Let's clear out our code and rename our new file to **0002_hello_world.py** and type the following code below then click Load/Save, Download Python Script and Flash.

```
from microbit import display


display.scroll('Hello World!')
```

Here we see the string *Hello World!* scroll across our micro:bit display. This is how we can work with output without having to use the REPL.

We see it only scroll once which is what we want. We will look at loops later in this course.

First we see *from microbit import display* which means there is a what we refer to as a *MicroPython module* which is nothing more than a *.py* MicroPython file or library of functions that help us make fun programs easily without having to do all of the low-level implementation to animate the screen in addition to all of the other pieces to make any functionality work.

We will get into more of what MicroPython modules are in later lessons.

We see *display* which is what we refer to as a *MicroPython* function which for our purposes represents a real-world object. In our case it is the display lights on the micro:bit. We see a dot . which means we are going to then extend our display object and make it do something. After the dot we see the world *scroll* which adds additional functionality to the display function to scroll the text.

Do not worry about fully understanding these concepts. I just wanted to bring them to your attention so if you had very basic questions about what each of these pieces mean you can at least get a very high-level understanding of what they are referred to. You do not at this stage have to understand how it all connects yet. This will take time and patience and will be an amazing journey for you which we will take together!

Let's clear out our code and rename our new file to **0003_hello_world_talk.py** and type the following code below then click Load/Save, Download Python Script and Flash.

```
from microbit import display, Image
from speech import say


SPEED = 95


display.show(Image.SURPRISED)
say('Hello World!', speed=SPEED)
display.show(Image.HAPPY)
```

Here we see our little micro:bit display a talking image then literally speak the string *Hello World!* and then display a smiling image.

This is now a third way to work with output with our micro:bit as we started off with text output in the REPL then we saw how our micro:bit could use the display to scroll output and finally here we can see how we can combine the display with speech functionality to allow it to talk to us.

Think of the amazing things you can do now just with this very basic toolset. The possibilities are unlimited and with each lesson we will keep learning and making this journey the most amazing ever!

Let's, on a very high level, review what is going on in the code so far. It is important to remember that we are at the beginning of our journey and like any journey the literal amount of new things we encounter all at once can be intimidating however we will develop these out so that when you continue to come across them in future lectures they will become more solidified in your understanding.

We see *from microbit import display, Image* which means let's take a walk into the microbit module (library) and go and grab the display book (function) and also before we leave the library we need to grab the Image book (class).

We then see *from speech import say* so let's walk across the street and go into the speech module (library) and checkout the say book (function).

Now we have all three books in hand, display, Image and say so we have the tools necessary to write our little app.

We then see a word all in caps called *SPEED* and it is assigned to the number *95*. This is what we call a constant or something that will remain the same and not change for the duration of our app. We make it in all CAPS to remind us that this will not change during the app however we can change the value in one place if we want to adjust rather than changing it all over the app.

We then see *display.show(Image.SURPRISED)* which simply displays the little talking image.

We then see *say('Hello World!')* which allows our little micro:bit to say the words Hello World! to us through it's little speaker.

Finally we see *display.show(Image.HAPPY)* which displays a happy face to us and will stay in that position until new code is flashed.

I know this is a good deal of new info for you but just spend a few minutes each day typing this into your MicroPython Web Editor and trying new strings to the REPL, display and to our micro:bit to speak back to you.

Learning is like exercise, the more you do the better you become at it.

Let's clear out our code and rename our new file to **0004_basic_io_repl.py** and type the following code below then click Load/Save, Download Python Script, Flash and Open Serial.

```
# We introduce the concept of a variable to which
# we reserve a little box in our computer's memory
# to hold the string which we are going to type
# when prompted to provide our favorite food and
# favorite drink
favorite_food = input('What is your favorite food? ')
favorite_drink = input('What is your favorite drink? ')

# Here we use MicroPython's built-in format method
# which is part of the string module's Formatter
# class as the following line of code will provide
# a response back based to the console based on
# our two variables which we inputted above
print('I love {0} and {1} as well!'.format(favorite_food, favorite_drink))
```

I want to introduce the concept of adding comments. We see a *#* and then everything after the *#* on a line is what we call a *comment*. These are helpful to remind us what is going on in our code.

When we start out we can use as many comments as we want. As we get more comfortable we will tend to use fewer comments as we will get a better handle of what is going on by looking at the Python code.

Earlier we saw the concept of a constant which holds a value that does not change when the app runs however now we are going to extend that concept to the variable.

A variable holds a value in those little boxes like we saw earlier and we can use this to store any information we want during our app's run. The difference here is that variables can change during our app and not stay constant.

We are also introducing the concept of basic input in MicroPython which we refer to as *input*. This is a built-in function like the *print* function that allows us to display a message in the REPL and then whatever we type will be then stored into the variable.

We see *favorite_food = input('What is your favorite food? ')* and all this does is display the words, What is your favorite food? and then allow us to type a string response and then it will be stored in *favorite_food*. For example if we typed *pizza* then the string *pizza* would be stored in the *favorite_food* variable.

We repeat the process for *favorite_drink* in the exact same way.

Finally we use the *print* function again and we use what we refer to as a *format method*. Notice we see *{0}* and *{1}* which are placeholders for our variables so what will happen is that if we used the word

pizza for *favorite_food* it would replace the *{0}* with *pizza* and if we used *Pepsi* for *favorite_drink* the *{1}* would be replaced with *Pepsi*.

Let's try it in the REPL! Let's click Open Serial.

```
What is your favorite food? pizza
What is your favorite drink? Pepsi
I love pizza and Pepsi as well!
MicroPython v1.13 on 2020-12-03; micro:bit v2.0.0-beta.2 with nRF52833
Type "help()" for more information.
>>>
```

It is now time for our first project!

Project 1 - Create a Candy Name Generator app - You are hired as a contract MicroPython Software Developer to help Mr. Willy Wonka rattle off whatever candy title he comes up with in addition to a flavor of that candy. When the program is complete, Mr. Willy Wonka will be able to type into the MicroPython REPL a candy title he dreams up in addition to the flavor of that candy. For example, Scrumpdiddlyumptious Strawberry.

STEP 1: Prepare Our Coding Environment

Let's clear out our code and rename our new file to **p_0001_candy_name_generator.py** and follow all of the steps we learned so far.

Give it your best shot and really spend some time on this so these concepts become stronger with you which will help you become a better MicroPython Developer in the future.

The real learning takes place here in the projects. This is where you can look back at what you learned and try to build something brand-new all on your own.

This is the hardest and more rewarding part of programming so do not be intimidated and give it your best!

If you have spent a few hours and are stuck you can find the solution here to help you review a solution. Look for the **Part_1_Basic_IO** folder and click on **p_0001_candy_name_generator.py** in GitHub.

https://github.com/mytechnotalent/Python-For-Kids

EXTRA CREDIT - Create a Talking Candy Name Generator app - If you are feeling adventurous, let's clear out our code and rename our new file

to **p_0001_candy_name_generator_ec.py** and follow all of the steps we learned so far.

Our task is to take the file we just created and add talking functionality to it.

Earlier in our lesson we learned how to program the micro:bit to talk so let's try to see if we an integrate that functionality into this app!

Please give it a few hours and if you do get stuck here is a solution to review. Look for the **Part_1_Basic_IO** folder and click on **p_0001_candy_name_generator_ec.py** in GitHub.

https://github.com/mytechnotalent/Python-For-Kids

I really admire you as you have stuck it out and made it to the end of your first step in the MicroPython Journey! Great job!

In our next lesson we will learn about MicroPython data types and numbers!

Chapter 10: DataTypes & Numbers

Today we are going to discuss datatypes and numbers as it relates to MicroPython on our micro:bit.

By the end of the lesson we will have accomplished the following.

```
* Written a 0005_calculator_repl.py app which will output add, subtract,
multiply and divide two numbers in the REPL or web terminal or console.

* Written a 0006_square_footage_repl app which will take a width and height in
feet from the repl and print and display the square footage in our micro:bit display LED
matrix.

* Written a 0007_final_score_talk_repl app which will calculate a final
score of a player and indicate the result of a hardcoded boolean.
```

We will focus on 4 primary primitive datatypes that are built-in to MicroPython.

```
* string
* integer
* float
* boolean
```

string

We are familiar with the concept of the string from last lesson. A string is nothing more than a string of characters.

Let's open up our Python Web Editor (full instructions were in Part 1 if you need to double-check) and type the following.

```
name = 'Kevin'
print(name)


Kevin
```

We see that our name variable properly prints the word *'Kevin'*.

Let's demonstrate that a *string* really is a string of characters. Each letter or character in a string is referred to as an *element*. Elements in MicroPython are what we refer to as *zero-indexed* meaning that the first *element* starts at *0* not *1*.

Let's try an example:

```
name = 'Kevin'
print(name[0])
print(name[1])
print(name[2])
print(name[3])
print(name[4])

print(name[-1])

print(name[1:4])

print(name[1:])


K
e
v
i
n

n

evi

evin
```

We see that we do have 5 characters starting at *0* and ending with *4* so the first *element* is *0* and the fifth *element* is *4*. We can also see that the *-1* allows us to get the last *element*. In addition we can see *1:4* prints the 2nd, 3rd and 4th *element* (1, 2, 3) but not the 5th *element*. We see that if we do *1:* that will print the 2nd *element* and the remaining elements.

If we try print an element that is out of bounds we will get an *IndexError*.

```
name = 'Kevin'
print(name[5])


IndexError: str index out of range
```

A *string* is what we refer to as *immutable* as you can't change individual letters or characters in a string however you can change the entire string to something else if it is a variable.

Let's look at an example to illustrate.

```
# CAN'T DO
name = 'Kevin'
name[0] = 'L'

TypeError: 'str' object doesn't support item assignment

# CAN DO
name = 'Kevin'
print(name)
name = 'Levin'
print(name)


Kevin
Levin
```

We can see that we in fact can't change an individual element in a *string* but we can change the *string* to be something else completely by reassigning the variable *name*.

When you add strings together you concatenate them rather than add them. Let's see what happens when we add two strings that are numbers.

```
print('1' + '2')


12
```

We can see we get the *string '12'* which are not numbers as they are strings concatenated together.

integer

An *integer* or *int* is a whole number without decimal places.

```
print(1 + 2)


3
```

Here we see something that we would naturally expect. When we add two integers together we get another integer as shown above.

float

A *float* is a number with fractions or decimals. One thing to remember about a *float* is that if you add, multiply, subtract or divide an *integer* with a *float* the result will ALWAYS be a *float*.

```
print(10.2 + 2)


12.2
```

boolean

A *boolean* has only two values which are either *True* or *False*. Make sure you keep note that you must use a capital letter at the beginning of each word.

A *True* value means anything that is not *0* or *None* and a *False* value is *None* or *0*.

The *None* keyword is used to define a null value, or no value at all. *None* is not the same as *0*, *False*, or an *empty string*.

None is a datatype of its own (*NoneType*) and only *None* can be *None*.

```
is_happy = True
print(is_happy)

is_angry = False
print(is_angry)

score = None
print(score)


True
False
None
```

A *boolean* can be used in so many powerful way such as setting an initial condition in an app or changing conditions based on other conditions, etc.

Type Checking & Type Conversion

We can check the datatype very easily by doing the following.

```
my_string = 'Kevin'
print(type(my_string))

my_int = 42
print(type(my_int))

my_float = 77.7
print(type(my_float))
```

```
my_boolean = True
print(type(my_boolean))


<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

You can also convert datatypes by what we refer to as casting or changing one datatype to another.

```
my_int = 42
print(type(my_int))


<class 'int'>
<class 'str'>
```

Math Operations In MicroPython

In MicroPython we have an order of operations that are as follows.

```
Parentheses ()
Exponents **
Multiplication *
Division /
Addition +
Subtraction -
```

If you look at them together you can see we have *PEMDAS*. This is a way we can remember.

In addition *multiplication* and *division* are of equal weight and the calculation which is left-most will be prioritized. This is the same for *addition* and *subtraction*.

```
print(5 * (9 + 5) / 3 - 3)

# First: (9 + 5) = 14
# Second: 5 * 14 = 70
# Third: 70 / 3 = 23.33334
# Fourth: 23.33334 - 3 = 20.33334


20.33334
```

APP 1

Let's create our first app for the day and call it **0005_calculator_repl.py**.

```
first_number = int(input('Enter First Number: '))
second_number = int(input('Enter Second Number: '))

my_addition = first_number + second_number
my_subtraction = first_number - second_number
my_multiplication = first_number * second_number
my_division = first_number / second_number

print('Addition = {0}'.format(my_addition))
print('Subtraction = {0}'.format(my_subtraction))
print('Multiplication = {0}'.format(my_multiplication))
print('Division = {0}'.format(my_division))
print(type(my_division))


Enter First Number: 3
Enter Second Number: 3
Addition = 6
Subtraction = 0
Multiplication = 9
Division = 1.0
<class 'float'>
```

Notice when we use division in MicroPython that the result is a float. This will always be the case.

APP 2

Let's create our second app for the day and call it **0006_square_footage_repl.py**:

```
from microbit import display

length = float(input('Enter length: '))
width = float(input('Enter width: '))

square_footage = length * width

print('Your room size is {0} square feet.'.format(square_footage))
display.scroll('Your room size is {0} square feet.'.format(square_footage))


Enter length: 4
Enter width: 5
Your room size is 20.0 square feet.
```

APP 3

Let's create our third app for the day and call it **0007_final_score_talk_repl.py**:

```
from microbit import display, Image
from speech import say

SPEED = 95

player_score = int(input('Enter Player Score: '))
player_score_bonus = int(input('Enter Player Score Bonus: '))
player_has_golden_ticket = True

player_final_score = player_score + player_score_bonus

display.show(Image.SURPRISED)
print('Player final score is {0} and has golden ticket is
{1}.'.format(player_final_score, player_has_golden_ticket))
say('Player final score is {0} and has golden ticket is {1}.'.format(player_final_score,
player_has_golden_ticket))
display.show(Image.HAPPY)


Enter Player Score: 4
Enter Player Score Bonus: 5
Player final score is 9 and has golden ticket is True.
```

Project 2 - Create a Talking Mad Libs app

Today we are going to create a talking mad libs app and call it **p_0002_talking_madlibs.py**:

Start out by thinking about the prior examples from today and spend an hour or two making a logical strategy based on what you have learned.

The app will first set the *SPEED* of the speech module. It will then get a noun from the user and then get a verb from the user and then finally create a madlib. Print this out and have the speech module talk out the result.

Give it your best shot and really spend some time on this so these concepts become stronger with you which will help you become a better MicroPython Developer in the future.

The real learning takes place here in the projects. This is where you can look back at what you learned and try to build something brand-new all on your own.

This is the hardest and more rewarding part of programming so do not be intimidated and give it your best!

If you have spent a few hours and are stuck you can find the solution here to help you review a solution. Look for the **Part_2_DataTypes_+_Numbers** folder and click on **p_0002_talking_madlibs.py** in GitHub.

https://github.com/mytechnotalent/Python-For-Kids

I really admire you as you have stuck it out and made it to the end of your second step in the MicroPython Journey! Great job!

In our next lesson we will learn about MicroPython conditional logic!

Chapter 11: Conditional Logic

Today we are going to discuss MicroPython conditional logic and application flow chart design for the micro:bit.

By the end of the lesson we will have accomplished the following.

* Written a 0008_career_councelor_repl.py app which will ask some basic questions and suggest a potential Software Engineering career path in the REPL.

* Written a 0009_heads_or_tails_game app which have us press either the A or B button to choose heads or tails and have our micro:bit randomize a coin toss and display the result in the LED matrix.

One of the most important parts of good Software Engineering is to take a moment and think about what it is you are designing rather than just diving in and beginning to code something.

We are very early into our most amazing journey and it is very important at this stage to develop good design patterns and procedures so that we can scale our amazing creations as we develop our skills!

To design any app we should first make a flow chart. In this course we will use the FREE draw.io online app to design our projects however feel free to use pen and paper. Either will be just fine.

Here is a link to **draw.io** that we will be using.

https://app.diagrams.net

When developing an app one of the most fundamental tools that we will need is an ability to allow the user to make choices. Once a user has made a choice we then want our app to do something specific based on their selection.

Conditional Logic

Literally everything in Computer Engineering is based on conditional logic. I would like to share these two videos by Computerphile that goes over the very core of logic gates in the machine code of all microcontrollers and computers.

https://youtu.be/UvI-AMAtrvE
https://youtu.be/VPw9vPN-3ac

```
AND
---
INPUT  OUTPUT
A      B      A AND B
0      0      0
0      1      0
1      0      0
1      1      1

OR
--
INPUT  OUTPUT
A      B      A OR B
0      0      0
0      1      1
1      0      1
1      1      1

NOT
---
INPUT   OUTPUT
A      NOT A
0      1
1      0

XOR
---
INPUT  OUTPUT
A      B      A XOR B
0      0      0
0      1      1
1      0      1
1      1      0
```

In MicroPython we have what we refer to as if/then logic or conditional logic that we can use and add to our tool box.

Here is some basic logic that will help demonstrate the point.

```
if something_a_is_true:
    do_something_specific_based_on_a
elif something_b_is_true:
    do_something_specific_based_on_b
else:
    do_something_that_is_default
```

The above is not runnable code it is referred to as pseudo code which is very important when we are trying to think about concepts.

The above has some rather long variable names which we would not necessarily have as long when we write our actual code however when

we pseudo code there are no rules and we can do what is natural for us to help us to better visualize and idea generate our process.

I want to discuss the concept of *Truthy* and *Falsey*. In conditional logic we have either *True* or *False*.

In addition to these two absolutes we have four values that when you apply conditional logic to they will be considered False.

These four conditions are *None*, *''*, *0* and *False*.

Let's review the following code to better understand.

```
my_none = None
my_empty_quotes = ''
my_zero = 0
my_false = False

if my_none:
    print('I will never print this line.')
elif my_empty_quotes:
    print('I will never print this line.')
elif my_zero:
    print('I will never print this line.')
elif my_false:
    print('I will never print this line.')
else:
    print('All of the above are falsey.')


All of the above are falsey.
```

The above is short-hand we can use in MicroPython it is doing the exact same thing as below however it is more readable above.

```
my_none = None
my_empty_quotes = ''
my_zero = 0
my_false = False

if my_none == True:
    print('I will never print this line.')
elif my_empty_quotes == True:
    print('I will never print this line.')
elif my_zero == True:
    print('I will never print this line.')
elif my_false == True:
    print('I will never print this line.')
else:
    print('All of the above are falsey.')
```

```
All of the above are falsey.
```

We can also utilize the NOT operator as well to make the opposite of the above.

```
my_none = None
my_empty_quotes = ''
my_zero = 0
my_false = False

if not my_none:
    print('I will print this line.')
if not my_empty_quotes:
    print('I will print this line.')
if not my_zero:
    print('I will print this line.')
if not my_false:
    print('I will print this line.')
else:
    print('I will never print this line.')


I will print this line.
I will print this line.
I will print this line.
I will print this line.
```

The above is short-hand we can use in MicroPython it is doing the exact same thing as below however it is more readable above.

```
my_none = None
my_empty_quotes = ''
my_zero = 0
my_false = False

if not my_none == True:
    print('I will print this line.')
if not my_empty_quotes == True:
    print('I will print this line.')
if not my_zero == True:
    print('I will print this line.')
if not my_false == True:
    print('I will print this line.')
else:
    print('I will never print this line.')


I will print this line.
I will print this line.
I will print this line.
I will print this line.
```

Outside of those conditions if a variable has something in it is considered *Truthy*.

```
my_empty_space = ' '
my_name = 'Kevin'
my_number = 42
my_true = True

if my_empty_space:
    print('I will print this line.')
if my_name:
    print('I will print this line.')
if my_number:
    print('I will print this line.')
if my_true:
    print('I will print this line.')
else:
    print('I will never print this line.')


I will print this line.
I will print this line.
I will print this line.
I will print this line.
```

Conversely we have the following with the NOT operator.

```
my_empty_space = ' '
my_name = 'Kevin'
my_number = 42
my_true = True

if not my_empty_space:
    print('I will never print this line.')
if not my_name:
    print('I will never print this line.')
if not my_number:
    print('I will never print this line.')
if not my_true:
    print('I will never print this line.')
else:
    print('All of the above are truthy.')


All of the above are truthy.
```

APP 1

Let's create our first app and call it **0008_career_councelor_repl.py**:

In our last two lessons we would normally dive into direct coding however now we are going to take our next steps toward good software design and create our first flow chart!

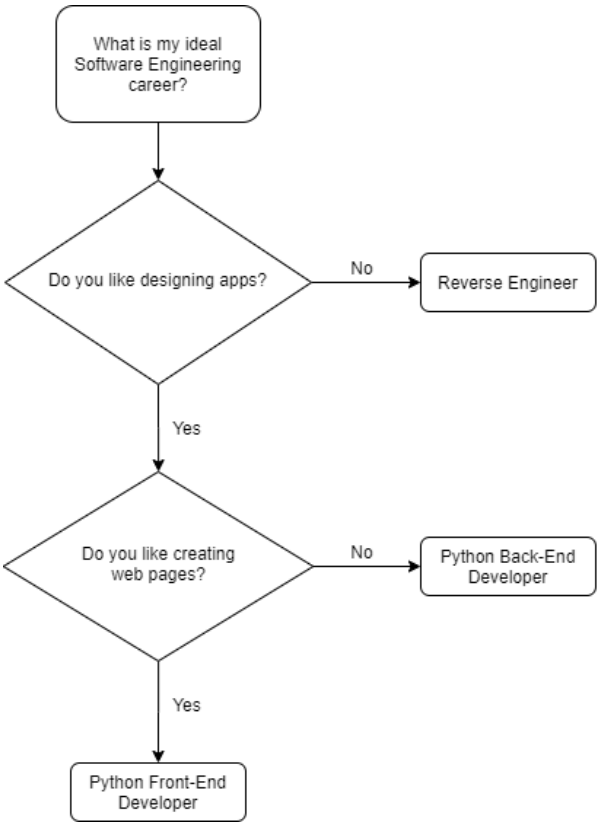
In this app we will ask two potential basic questions of the user and based on their responses we will make a suggestion on a potential Software Engineering career path to explore.

Let's open **draw.io** and start designing!

When we visit the site we see an option to create a flow chart. Let's select that option and click Create then it will prompt us to save our design. Let's call it **0008_career_councelor_repl** and it will save the file with the **.xml** extension and load up a default pre-populated example which we will modify.

We start by defining a problem or by asking a question that we are looking to solve. In our case our design begins with a question which is 'What is my ideal Software Engineering career?'.

We are going to make this app very simple so it will not have all of the options that would be more practical however we are going to ask three basic questions and based on those answers we will suggest a Software Engineering career path.



As we can see above this is a very simple design as we only have three options but it is perfect for us to start thinking about how we might make a powerful design!

We start off with a rounded-rectangle where we define the purpose of our app. In our case we want to figure out the best Software Engineering career for us.

The diamond represents a decision that we need to make or have our app ask the user and based on the response will suggest a career option or continue to ask an additional question which based on that response will suggest one or another career path.

Now that we have a basic level design we can start coding!

```
print('What is my ideal Software Engineering career?\n')

like_designing_apps = input('Do you like designing apps? (\'y\' or \'n\'): ').lower()

if like_designing_apps == 'n':
    print('Research a Reverse Engineering career path.\n')
else:
    like_creating_web_pages = input('Do you like creating web pages? (\'y\' or \'n\'):
').lower()
    if like_creating_web_pages == 'y':
        print('Research a Python Front-End Developer career path.\n')
    else:
        print('Research a Python Back-End Developer career path.\n')
```

We first greet the user with a larger conceptual question. We then use the newline character to make a new blank line for our code to be more readable.

We then create a variable where we ask the user a question and based on the response will either suggest a career path or ask another question.

If the answer is not '*n*' then we will ask another question and if that answer is '*y*' then we make a suggestion otherwise we make another suggestion.

In this very simple example we do not check for bad responses meaning anything other than a '*y*' or '*n*' as I did not want to over complicate our early development. As we progress we will build more robust solutions that will account for accidental or improper input.

In this simple example we could have very easily designed this without the flow chart but what if our logic was more robust?

Taking the time to design a flow chart in **draw.io** or on paper is a good Software Engineering design methodology to use as you progress in your journey no matter if you want to use Software Engineering to teach, create or do it as your career.

APP 2

Let's create our second app and call it **0009_heads_or_tails_game.py**:

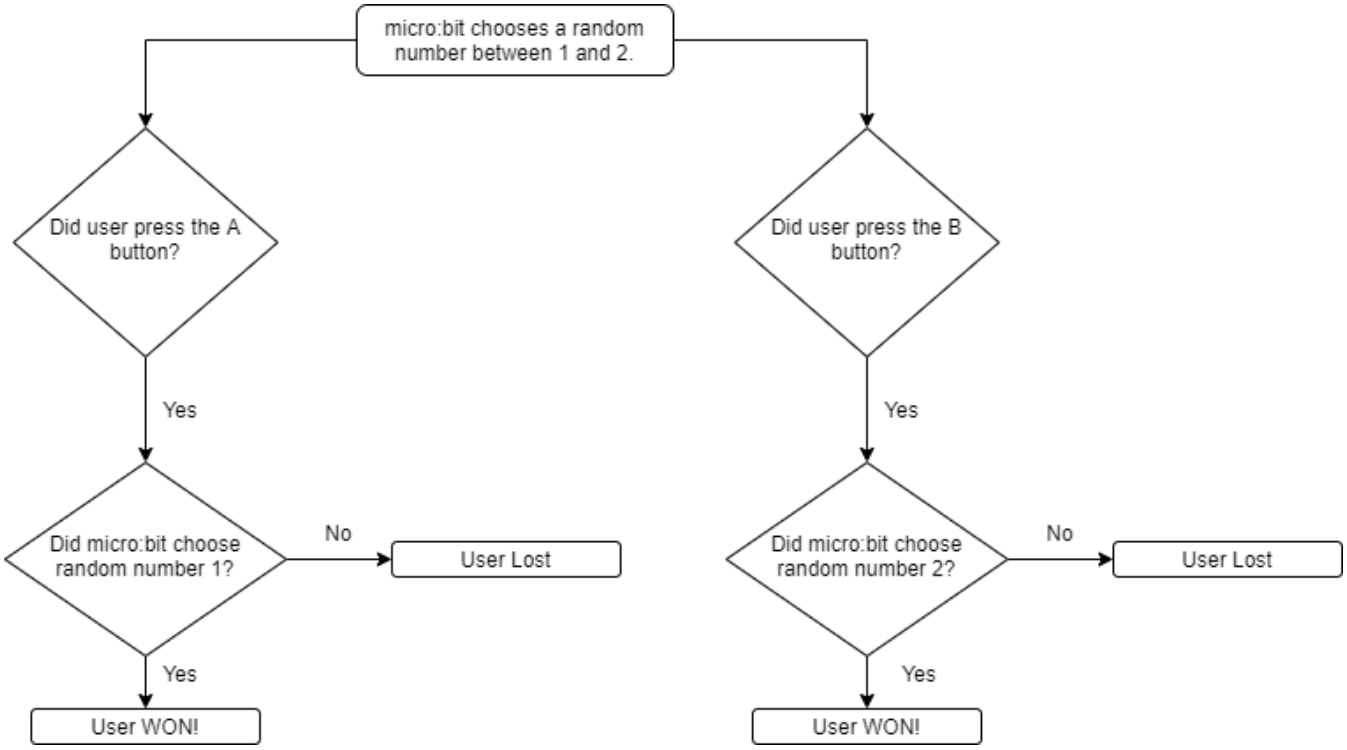
In this app we are going to introduce the random module. The random module allows MicroPython to pick a random number between a series of values. In our case we are going to choose a random number between *1* and *2*.

If our micro:bit chooses 1 that will be heads and if it chooses 2 it will be tails.

After our micro:bit has made a choice it will then prompt the user to press either *A* or *B* to guess *heads* or *tails*.

If the user guessed the selection which the micro:bit chose they win otherwise they lost.

Let's make a flow chart to design this logic.



Based on our flow chart, let's code!

```
from random import randint
from microbit import display, button_a, button_b

display.scroll('Heads = A')
display.scroll('Tails = B')
random_number = randint(1, 2)

while True:
    if button_a.is_pressed():
        if random_number == 1:
            display.scroll('You WON!')
            break
        else:
            display.scroll('You Lost')
            break

    if button_b.is_pressed():
        if random_number == 2:
            display.scroll('You WON!')
            break
        else:
            display.scroll('You Lost')
            break
```

Here we import the *random* module. We are going to use the *randint* function where we put in a range of numbers to pick from. In our case we are asking the micro:bit to choose a random number between *1* and *2* and will scroll some basic instructions.

We use the *while* loop to start our game loop and await the player to press either *A* or *B*. Upon the selection we scroll either a win or lose and then break out of the infinite loop.

Comparison Operators

In addition we can use comparison operators to check for the following.

```
>  greater than
<  less than
>= greater than or equal to
<= less than or equal to
== equal to
!= not equal to
```

Let's work with some examples.

```
number = 4

if number <= 5:
    print('Number is less than 5.\n')
elif number < 8:
    print('Number is less than 8 but not less than or equal to 5.\n')
else:
    print('Number is greater than or equal to 8.\n')
```

Let's talk through our logic. We first define a *number* as *4*. Our first conditional checks to see if the number is less than or equal to *5*. This is the case so that line will print.

```
Number is less than 5.
```

Now let's make the number 8.

```
number = 8

if number <= 5:
    print('Number is less than 5.\n')
elif number < 8:
    print('Number is less than 8 but not less than or equal to 5.\n')
else:
    print('Number is greater than or equal to 8.\n')
```

We can see that the number will fall into the else block as because it is does not fall under the first or second condition.

```
Number is greater than or equal to 8.
```

I have deliberately not discussed the *if* vs *elif* conditionals until now so that I can really illustrate a point.

When we use if and there are other if statements in the conditional it will continue to check regardless if the first condition was met. Let's look at the same example however modified to two *if* statements rather than an *if* and *elif* statement.

```
number = 4

if number <= 5:
    print('Number is less than 5.\n')
if number < 8:
    print('Number is less than 8 but not less than or equal to 5.\n')
else:
    print('Number is greater than or equal to 8.\n')
```

Here we see the first two conditions met.

```
Number is less than 5.

Number is less than 8 but not less than or equal to 5.
```

Is this what you expected? Let's think about it logically. When we say elif we mean else if rather than simply if. This means if the condition in the elif has been met, do not go any further otherwise keep checking.

Project 3 - Create a Talking Number Guessing Game app

Now it is time for our project! This is going to be a FUN one as we are going to use the talking features we have used in our last two lessons and include all of the following features.

Let's call name it **p_0003_talking_number_guessing_game.py**

- Create a flow chart to diagram our app.
- Create a SPEED constant and initialize it to 95.
- Create a random number generator and have the micro:bit choose a random number between *1* and *9* by creating a *random_number* variable while adding functionality within the *random* module to use the *randint* method.
- Create a *number_position* variable and initialize it to 1.
- Create the I*mage.SURPRISED* when the micro:bit speaks and *Image.HAPPY* when it is done speaking like we did in our prior lessons and have it say, *'Pick a number between 1 and 9'*. Remember to use the *SPEED* constant as an argument in the *say* function.
- Create a *while True:* loop and put the rest of the code that you are about to create within its scope.
- Create a *display.show()* to display the *number_position*.
- Create conditional logic within button_a such that if the *number_position* is equal to *9* then pass so that this will prevent the user to advance beyond *9*.
- Create conditional logic within button_a such that if you press *button_a* at this point will advance the numbers up to and including *9* and update the display with the current *number_position*.
- Create conditional logic within *button_b* such that if the *number_position* is equal to *1* then *pass* so that this will prevent the player to decrease below *1*.
- Create conditional logic within *button_b* such that if you press *button_b* at this point will decrease the numbers up to and including *1* and update the display with the current *number_position*.

- When the user has the display showing the number they want to select as an answer, create logic so that when the user press the *logo* it will lock in the the user's selection.
- Create conditional logic within the *pin_logo* such that if the *number_position* is equal to the *random_number* then they are correct and have the micro:bit do its talking sequence as outlined in above with the *Image.SURPRISED* and *Image.HAPPY* while having it speak, *'Correct'* and then break out of the *while* loop to end the game.
- If the answer is not correct have the micro:bit do the same sequence however have it say, *'The number I chose is {0}'* to display the winning *random_numbe*r and then break out of the *while* loop to end the game.

This is a very tough challenge and should take several hours. I do not want you to get discouraged here but rather take this as an inspirational exercise to really push you to stretch your muscles and grow as a Developer.

A Developer utilizes Google for asking questions, reaching out to a mentor/peer to discuss ideas in addition to reading the official documentation of the language they are using in addition to reviewing other code examples on the web.

I would like you to visit the micro:bit V2 docs and get familiar with the site.

https://microbit-micropython.readthedocs.io/en/v2-docs

Here I would like you to review the Input/Output pins. You will use the *pin_logo.is_touched()* in your app.

I also want you to think about adding a *number_position = 1* to start out with so that you have a starting point for your game.

I also want you to make sure you use the break keyword when a user has touched the *pin_logo.is_touched()* so that it can break out of the while loop and end the game.

Finally I will give you some starter logic for *button_a*.

We see something new here. We see += together and that is short hand for *number_position = number_position + 1*.

```
    if button_a.was_pressed():
        if number_position == 9:
            pass
        else:
            number_position += 1
            display.scroll(number_position)
```

Think about how *button_b* might be implemented.

If after several hours or days and you get stuck you can find my solution below. Look for the **Part_3_Conditional_Logic** folder and click on **p_0003_talking_number_guessing_game.py** in GitHub.

https://github.com/mytechnotalent/Python-For-Kids

I really admire you as you have stuck it out and made it to the end of your third step in the MicroPython Journey! Today was particularly challenging! Great job!

In our next lesson we will learn about lists, dictionaries and loops!

Chapter 12: Lists, Tuples, Dictionaries & Loops

Today we are going to learn about the powerful data structures called lists and dictionaries in addition to looping logic.

By the end of the lesson we will have accomplished the following.

```
* Written a 0010_rock_paper_scissors_repl app which will take a simple list and create a FUN game where you play against the computer working with the random module.

* Written a 0011_journal_repl app which will take a dictionary of journal entries and have the ability to add new entries.

* Written a 0012_high_score_repl app which will you enter in a list of scores and use a for loop to find the highest score.
```

Lists

Unlike variables which store only one piece of data a list can store many items or pieces of data that have a connection with each other.

Lists are often called arrays in other languages.

Imagine you wanted to store all of the letters of the alphabet. If you had individual variables you would have to make 26 different variables. With a list you can use one variable and store an array or collection of each of the letters.

Let's look at an example of a list in the repl.

```
chocolates = ['caramel', 'dark', 'milk']

print(chocolates)
print(chocolates[0])
print(chocolates[1])
print(chocolates[-1])
print(chocolates[-2])
print(chocolates[:])
print(chocolates[:-1])
print(chocolates[1:])

chocolates.append('sweet')

print(chocolates)

chocolates.remove('milk')

print(chocolates)
```

```
['caramel', 'dark', 'milk']
caramel
dark
milk
dark
['caramel', 'dark', 'milk']
['caramel', 'dark']
['dark', 'milk']
['caramel', 'dark', 'milk', 'sweet']
['caramel', 'dark', 'sweet']
```

Wow! WOOHOO! Look at all that chocolate! Let's break down exactly what is happening.

We first create list of chocolates and have 3 elements in it. Unlike strings, lists are mutable or changeable so we can keep on adding chocolate! How cool is that?

We then print the entire list of chocolates where it will print each chocolate on a separate line.

We then add a new chocolate, *'sweet'*, to the list.

We then print the updated list.

We then remove a chocolate, *'milk'*, from the list.

We then print the updated list.

In the last chapter we discussed the importance of flow charts to design our work. Going forward we will work with what we call an Application Requirements document which is a written step of TODO items.

APP 1

Let's create our first app and call it **0010_rock_paper_scissors_repl**:

Let's create our Rock Paper Scissors Application Requirements document and call it **0010_rock_paper_scissors_ar**:

```
Rock Paper Scissors Application Requirements
-------------------------------------------
1. Define the purpose of the application.
   a. Create a game where a computer randomly chooses a number between
      0 and two and we choose a selection of either rock, paper or
      scissors.  Based on the rules either the computer or the player
      will win.
2. Define the rules of the application.
   a. Rock wins against scissors.
   b. Scissors wins against paper.
   c. Paper wins against rock.
3. Define the logical steps of the application.
   a. Create a game_choices list and populate it with the 3 str choices.
   b. Create a random number between 0 and 2 and assign that into a
      computer_choice variable.
   c. Create player_choice variable and get an input from the player
      and cast to an int with a brief message.
   d. Create conditional logic to cast the int values into strings
      for both the computer and player.
   e. Display computer choice and player choice.
   f. Create conditional logic to select a winner based on rules
      and print winner.
```

Let's create our app based on the above criteria.

```
import random

game_choices = ['Rock', 'Paper', 'Scissors']

computer_choice = random.randint(0, 2)

player_choice = int(input('What do you choose?  Type 0 for Rock, 1 for Paper, 2 for
Scissors. '))

if computer_choice == 0:
    computer_choice = 'Rock'
elif computer_choice == 1:
    computer_choice = 'Paper'
else:
    computer_choice = 'Scissors'
if player_choice == 0:
    player_choice = 'Rock'
elif player_choice == 1:
    player_choice = 'Paper'
else:
    player_choice = 'Scissors'

print('Computer chose {0} and player chose {1}.'.format(computer_choice, player_choice))

if computer_choice == 'Rock' and player_choice == 'Scissors':
  print('Computer - {0}'.format(game_choices[0]))
  print('Player - {0}'.format(game_choices[2]))
  print('Computer Wins!')
elif computer_choice == 'Scissors' and player_choice == 'Rock':
```

```
  print('Computer - {0}'.format(game_choices[2]))
  print('Player - {0}'.format(game_choices[0]))
  print('Player Wins!')
elif computer_choice == 'Scissors' and player_choice == 'Paper':
  print('Computer - {0}'.format(game_choices[2]))
  print('Player - {0}'.format(game_choices[1]))
  print('Computer Wins!')
elif computer_choice == 'Paper' and player_choice == 'Scissors':
  print('Computer - {0}'.format(game_choices[1]))
  print('Player - {0}'.format(game_choices[2]))
  print('Player Wins!')
elif computer_choice == 'Paper' and player_choice == 'Rock':
  print('Computer - {0}'.format(game_choices[1]))
  print('Player - {0}'.format(game_choices[0]))
  print('Computer Wins!')
elif computer_choice == 'Rock' and player_choice == 'Paper':
  print('Computer - {0}'.format(game_choices[0]))
  print('Player - {0}'.format(game_choices[1]))
  print('Player Wins!')
else:
  if computer_choice == 'Rock':
    print('Computer - {0}'.format(game_choices[0]))
    print('Player - {0}'.format(game_choices[0]))
  elif computer_choice == 'Paper':
    print('Computer - {0}'.format(game_choices[1]))
    print('Player - {0}'.format(game_choices[1]))
  else:
    print('Computer - {0}'.format(game_choices[2]))
    print('Player - {0}'.format(game_choices[2]))
  print('Draw!')
```

Woah! That is a lot of code! No worry! We are Pythonista's now and we shall be victorious!

We start by creating a *list* of three *str* items.

We then pick a random number between *0* and *2* and assign to the computer.

We prompt the player for a number between *0* and *2* representing the choices.

We then create conditional logic (ONE OF OUR MICROPYTHON SUPERPOWERS) to convert all computer numbers into *str* logic.

We then print the results.

We then create more conditional logic to figure out and decide a winner.

Tuples

A Python t*uple* is just like a list but it is *immutable*. You would use this if you wanted to create a list of constants that you do not want to change.

```
RED = (255, 0, 0)
```

The elements such as *RED[0]* will be *255* but you can't reassign it.

Dictionaries

A Python dictionary is what we refer to as a key/value pair.

During one of the most amazing events I ever took part of was called the micro:bit LIVE 2020 Virtual to which I presented an educational app called the Study Buddy. The Study Buddy gave birth to the concept of an Electronic Educational Engagement Tool designed to help students learn a new classroom subject with the assistance of a micro:bit TED (Talking Educational Database) and a micro:bit TEQ (Talking Educational Quiz).

Below is a video showing the POWER of such an amazing tool!

https://youtu.be/0OG5Vfdh5bM

This concept would have never been possible without a Python dictionary!

We started out with a simple key/value pair called a *generic_ted* or *Talking Educational Database* like the following.

```
generic_ted = {
                'your name': 'My name is Mr. George.',
                'food': 'I like pizza.',
              }
```

We have a *str* key called *'your name'* and a *str* value of *'My name is Mr. George.'*, which we can retrieve a value by doing the following by adding to the code above.

```
print(generic_ted['your name'])


My name is Mr. George.
```

Dictionaries are also *mutable* which means you can add to them! Let's add to the code above.

```
generic_ted['drink'] = 'Milkshake'

print(generic_ted)


{'your name': 'My name is Mr. George.', 'drink': 'Milkshake', 'food': 'I like pizza.'}
```

We see the structure come back as a list of key/value pairs. This is an unordered datatype however we see above how we can get the value of a particular key and we see how to create a new value as well.

We can also nest lists inside of a dictionary as well. Let's add to the code above.

```
generic_ted['interests'] = ['Python', 'Hiking']

print(generic_ted)


{'your name': 'My name is Mr. George.', 'drink': 'Milkshake', 'interests': ['Python',
'Hiking'], 'food': 'I like pizza.'}
```

You can see that we do have a list as a value in the *interests* key.

We can also remove an item from a dictionary. Let's add to the code above.

```
generic_ted.pop('drink')

print(generic_ted)


{'your name': 'My name is Mr. George.', 'interests': ['Python', 'Hiking'], 'food': 'I
like pizza.'}
```

We see that the *'drink'* key and *'Milkshake'* value is now removed from our dictionary.

One thing to remember about dictionaries is that the key MUST be unique in a given dictionary.

APP 2

Let's create our second app and call it **0011_journal_repl**:

Let's create our Journal Application Requirements document and call it **0011_journal_ar**:

```
Journal Application Requirements
--------------------------------
1. Define the purpose of the application.
   a. Create a Journal application where we have a starting dictionary of
      entries and create the ability to add new ones.
2. Define the logical steps of the application.
   a. Create a journal dictionary and populate it with some starting values.
   b. Create a new empty dictionary called new_journal_entry.
   c. Create a new entry and date in the new_journal_entry.
   d. Append the new_journal_entry dictionary into the journal dictionary.
   e. Print the new results of the appended journal dictionary.
```

Let's create our app based on the above criteria.

```
journal = [
    {
        'entry': 'Today I started an amazing new journey with MicroPython!',
        'date': '12/15/20',
    },
    {
        'entry': 'Today I created my first app!',
        'date': '12/16/20',
    }
]

new_journal_entry = {}

new_journal_entry['entry'] = 'Today I created my first dictionary!'
new_journal_entry['date'] = '01/15/21'
journal.append(new_journal_entry)

print(journal)


[{'date': '12/15/20', 'entry': 'Today I started an amazing new journey with
MicroPython!'}, {'date': '12/16/20', 'entry': 'Today I created my first app!'}, {'date':
'01/15/21', 'entry': 'Today I created my first dictionary!'}]
```

Very cool! Let's break this down.

We first create a *journal* dictionary and populate it.

We then create a *new_journal_entry* dictionary which is an empty dictionary.

We then add values into the *new_journal_entry*.

We then append the *new_journal_entry* dictionary into the *journal* dictionary.

We then print the new appended *journal* dictionary.

Loops

This is so much fun! Now it is time to REALLY jazz up our applications! In MicroPython we have a *for* loop which allows us to iterate over a list of items and then do something with each item and we also have the ability to take a variable and iterate over it with a range of numbers and finally we have the *while* loop which allows us to do something while a condition is true.

Woah that is a lot to process! No worries! We will take it step-by-step!

Let us start with a *for* loop that allows us to iterate over an item and then do something with each iteration. Let's go back to our YUMMY chocolates!

```
chocolates = ['caramel', 'dark', 'milk']

for chocolate in chocolates:
    print(chocolate)


caramel
dark
milk
```

WOW! WOOHOO! We do not have to use three print lines anymore only one! WOOHOO! Does a dance! LOL!

We can begin to see the POWER of such a concept as we could add other things to do for each chocolate as well!

```
chocolates = ['caramel', 'dark', 'milk']

for chocolate in chocolates:
    print(chocolate)
    print('I am eating {0} chocolate!  YUMMY!'.format(chocolate))


caramel
I am eating caramel chocolate!  YUMMY!
dark
I am eating dark chocolate!  YUMMY!
milk
I am eating milk chocolate!  YUMMY!
```

We can see that we printed the individual chocolate iteration and then we ate it! YAY!

We also have the ability to take a variable and iterate over it with a range of numbers.

```
for number in range(1, 5):
  print(number)

print()

for number in range(3, 9, 3):
  print(number)


1
2
3
4

3
6
```

Here we printed an int number variable and started from 1 and went up to but NOT including the last number *5*. We printed *1*, *2*, *3*, *4* as shown.

We can also start at another number *3* and go up to but NOT include *9* so *3* to *8* and print every 3rd number which would be *3* and *6*.

We also have the *while* loop which allows us to do a series of things while something is true.

```
has_chocolate = True

while has_chocolate:
    print('I have chocolate!')
    print('Yay I just ate it!')
    print('So yummy!')
    has_chocolate = False

print('Oh no I ate all my chocolate! :(')


I have chocolate!
Yay I just ate it!
So yummy!
Oh no I ate all my chocolate! :(
```

We can see that we start with a boolean flag called *has_chocolate* which is *True*.

We then enter the *while* loop because and ONLY because *has_chocolate* is *True*.

We then print a line, print another line and print another line and then set the flag *False* causing the while loop to break.

This won't make a good deal of sense yet until we start to get into functions where we can really see the power of such a loop!

APP 3

Let's create our third app and call it **0012_high_score_repl**:

Let's create our High Score Application Requirements document and call it **0012_high_score_ar**:

```
High Score Application Requirements
-----------------------------------
1. Define the purpose of the application.
   a. Create a High Score application where you enter in a list of scores and use
      a for loop to find the highest score
2. Define the logical steps of the application.
   a. Create a str scores variable and input getting a list of numbers
      separated by a space and when the user presses enter it will
      split each number based on the space into a scores list.
   b. Cast the entire list from a str list into an int list using a for
      loop.
   c. Print the new int list.
   d. Create a highest_score variable and init to 0.
   e. Create a for loop where we iterate over each score and create a
      conditional to check if each score is greater than the hightest
      score and if it is then take that score value and assign it
      to the highest_score variable and keep iterating.  If we find
      another score that is bigger than the highest_score than
      assign that new score to highest_score.
   f. Print highest_score var.
```

Let's create our app based on the above criteria.

```
# We need to make sure we are entering in a str to avoid a
# TypeError: can't convert list to int
scores = input('Input each score separated by a space: ').split()

# Convert str list into an int list
for n in range(0, len(scores)):
  scores[n] = int(scores[n])

print(scores)

highest_score = 0

for score in scores:
  if score > highest_score:
    highest_score = score
```

```
print('The highest score is {0}!'.format(highest_score))
```

Project 4 - Create a Talking Caramel Chocolate Adventure Game app

Now it is time for our project! This is going to be a FUN one as we are going to create an adventure game where we use dictionaries, lists and loops to find the hidden caramel chocolate!

Let's call name it **p_0004_talking_caramel_chocolate_adventure_game.py**

* Create a Talking Caramel Chocolate Adventure Game Application Requirements document.
* Create a *SPEED* constant and initialize it to *95*.
* Create a *chocolates* list and populate it with *milk*, *white*, *dark*, *caramel* and *mint* strings.
* Create an empty *rooms* dictionary.
* Create a *room_number* variable and initialize it to *1*.
* Create a *guesses* variable and initialize it to *2*.
* Create a *for* loop and iterate a room variable in *range(1, len(chocolates) + 1* as we have to remember that our range does not include the final number. Within the scope of the for loop create a *random_chocolate* variable and assign into it a *choice(chocolates)* to which choice is part of the random module so you have to import it and the argument is chocolates which is our list. This will randomly pick one of the 5 list items and assign into *random_chocolate*. Then we want to assign the *random_chocolate* variable into *rooms[room]* which will take the random value that the micro:bit chose and put it into the given room iteration. If we are going through the first iteration then room value will be *1*. If we are going through the second iteration the room value will be *2*, etc. We want to make sure we do not duplicate our chocolates so before we leave an iteration we have to *chocolates.remove(random_chocolate)* so that the *random_chocolate* value is unique in future iterations of the *for* loop.
* We want to create our talking sequence that we have used in the past to have our micro:bit speak *'Welcome to the Talking Caramel Chocolate Adventure Game!'* and add a *sleep(1*) to give the speaking some pause so that is more natural in its speaking progression. Remember to import time to use the sleep method.
* Create the same sequence however within it have two separate say statements which will be, *'Press the A and B buttons to move back and fourth through 5 '* and *'different rooms as your goal find the room with the caramel '* and *'chocolate.'* and add a *sleep(1)*.
* Create the same sequence however within it have it say statement which will be, *'You get two guesses.'* and add a *sleep(1)*.

* Create the same sequence however within it have two separate say statements which will be, *'If you press the logo and if the caramel chocolate ' and 'is in that room you win!'* and add a *sleep(1)*.
* Create the same sequence however within it have it say statement which will be, *'Let the games begin!'* and add a *sleep(1)*.
* Create a *while guesses > 0:* loop and put the rest of the code within its scope.
* Have the display show the *room_number*.
* Last week we learned how to create conditional logic to control *button_a* and *button_b* to display *1* through *9*. Do the same kind of conditional logic but work with *1* to *5*. Then display the *room_number*.
* Create conditional logic within the *pin_logo* such that if *rooms[room_number]* is equal to *'caramel'* create our talking sequence to say, *'You found the caramel chocolate! Great job!'* and break out of our *while* loop else have our sequence say, *'Sorry this room has {0} chocolate.'* and use the *rooms[room_number]* in our format function then decrease guesses by *1*.
* Finally outside of the *while* loop create conditional logic such that *if guesses <= 0:* show our talking sequence and have it say, *'Sorry about that. Please try again by click the reset button.' and sleep(1)* else have our talking sequence say, *'Click the reset button to play again.'* and *sleep(1)*.

This is a very tough challenge and you are putting together all of the things you have learned today. Think about running through all of the code you learned today over a few hours and then to make this final project will likely take 4-5 hours.

If after several hours or days and you get stuck you can find my solution below. Look for the **Part_4_Lists_Dictionaries_Loops** folder and click on **p_0004_talking_caramel_chocolate_adventure_game.py** and **p_0004_talking_caramel_chocolate_adventure_game_ar** in GitHub.

https://github.com/mytechnotalent/Python-For-Kids

I really admire you as you have stuck it out and made it to the end of your fourth step in the MicroPython Journey! Today was insanely challenging! Great job!

In our next lesson we will learn about functions!

Chapter 13: Functions

Today we are going to learn about functions.

By the end of the lesson we will have accomplished the following.

```
* Written a 0013_number_guessing_game_repl app where we guess a number between 1 and 9 much like our project in lesson 3 however in the repl and utilizing functions.
```

Functions

Functions allow for efficient code readability, portability, reusability and scalability for your MicroPython projects and development.

A function starts off with a *def* keyword followed by the function name then parenthesis and the function parameters and a final parenthesis and a semicolon.

```
def my_first_function(my_first_param):
    pass
```

The above would do nothing however would be valid. Keep in mind that a function does nothing unless you call it like the following.

```
my_first_function('some str')
```

A function also has a *return* value. In the above case there was no return value specified so it would default to *None*. We will see examples of this throughout this lesson.

There is some terminology to be aware of as well. The *my_first_param* in the function definition is referred to as a *parameter*. When you call a function, otherwise known as running it, the *'some str'* is referred to as an *argument*.

There are 4 types or variations of functions.

- no params, no return value (return value will default to None)
- params, no return value
- params, return value
- another function passed as a param (higher order function)

Let's first take a look at a function that has no params and no return value.

```
def add():
    a = 2
    b = 2
    c = a + b
    print(c)


add()


4
```

It is a PEP8 standard that functions have two blank spaces before and after them. PEP8 is the official Python Style Guide.

https://www.python.org/dev/peps/pep-0008

We can also prove that the return is *None* by the following.

```
return_value = add()
print(return_value)


4
None
```

Let's take a look at a function that has params but no return value.

```
def add(a, b):
    c = a + b
    print(c)


return_value = add(2, 2)


print(return_value)


4
None
```

There are also what we refer to as *default params* or *default arguments* that you can assign in the function definition.

```
def add(a, b, name='Kevin'):
    c = a + b
    print(c)
    print(name)


return_value = add(2, 2)

print(return_value)


4
Kevin
None
```

Take note that *'Kevin'* printed but was not mentioned in the function call.

You can also change a *default* param by doing the following.

```
def add(a, b, name='Kevin'):
    c = a + b
    print(c)
    print(name)


return_value = add(2, 2, 'Katie')

print(return_value)


4
Katie
None
```

Let's take a look at a function that has params and a return value.

```
def add(a, b):
    c = a + b
    return c


return_value = add(2, 2)

print(return_value)


4
```

Let's take a look at a function that passes another function as a param as it is called a *higher order function*.

```
def add(a, b):
    return a + b


def calculator(a, b, func):
    return func(a, b)


print(calculator(2, 2, add))


4
```

In our first app we will learn about docstrings and how to properly document each function we create so others can understand how to use them.

APP 1

Let's create our first app and call it **0013_number_guessing_game_repl**:

Let's create our Number Guessing Game Application Requirements document and call it **0013_number_guessing_game_ar**:

```
Number Guessing Game Application Requirements
---------------------------------------------
1. Define the purpose of the application.
   a. Create a game where a computer randomly chooses a number between
      1 and 9.  The player will enter a choice and will get 3 guesses
      and a hint with a short message indicating they are too high or
      too low if they do not guess the right number.
2. Define the rules of the application.
   a. If the player guesses the number they win.
   b. If the player does not guess the right number they get 3
      hints and if they get it they win otherwise lose.
3. Define the logical steps of the application.
   a. Create a turns_left var and init to 3.
   b. Create a guess_number function to obtain guess and return it.
   c. Create a did_win function with params f_guess, f_correct_answer and
      f_turns_left and it will return turns - 1 or a message to indicate a win.
   d. Create a text intro explaining the game and rules.
   e. Create an answer var and obtain a random number between 1 and 9.
   f. Create a guess var and init to 0.
   g. Create a game loop with logic to obtain a players guess and
      and logic to allow for a max of 3 turns and conditional
      logic to check if they have run out of turns then display a message
      they are out of turns and if there are turns left another message
```

```
      to guess again.
```

Let's create our app based on the above criteria.

```
from random import randint


def guess_number(f_guess, f_turns_left):
    """
    Functio to obtain player guess

    Params:
        f_guess: int
        f_turns_left: int

    Returns:
        int, str
    """
    try:
        if f_guess < 1 or f_guess > 9:
            raise ValueError
        return f_guess, f_turns_left - 1
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1
    except TypeError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1


def did_win(f_guess, f_correct_answer, f_turns_left):
    """
    Function to check player guess against the correct answer

    Params:
        f_guess: int
        f_correct_answer: int
        f_turns_left: int

    Returns:
        str, int, None
    """
    if f_turns_left >= 1:
        if f_guess > f_correct_answer:
            return 'HINT: Lower Than {0}'.format(f_guess), f_turns_left - 1
        elif f_guess < f_correct_answer:
            return 'HINT: Higher Than {0}'.format(f_guess), f_turns_left - 1
        else:
            return 'You won!', None


print('RULES: Guess a number between 1 and 9.')

correct_answer = randint(1, 9)
turns_left = 3
guess = 1
```

```
while guess != correct_answer:
    if turns_left >= 1:
        guess = input('Guess: ')
        guess_response, turns_left = guess_number(guess, turns_left)
        if turns_left > 1:
            print('{0} turns left.'.format(turns_left))
        elif turns_left == 1:
            print('{0} turn left.'.format(turns_left))
    else:
        print('The correct answer is {0}, let\'s play again!'.format(correct_answer))
        break

    if isinstance(guess_response, str):
        print(guess_response)
    else:
        if turns_left:
            game_status, turns_left = did_win(guess_response, correct_answer, turns_left)
            if game_status == 'You won!':
                print(game_status)
                break
            else:
                print(game_status)
```

This can be very overwhelming so let's carefully break this down.

We start out with a global *turns_left* var and set it to *3*.

We then create a *guess_number* function with docstrings and create a *try* and *except* block to properly handle incorrect input as we check for a value between *1* and *9* and return that value to the main loop. If the user enters something different we raise a *ValueError* and we return *False* to the main loop.

We then create a *did_win* function that takes 3 params which are *f_guess*, *f_correct_answer* and *f_turns_left* and we want to make sure our params do not shadow the names of the actual argument vars when we call it. We therefore prepend an *f* to the beginning of each var. If the *f_turns_left* is > *1* that means we have at least one more turn left therefore enter into the next *if* block to check if the guess is too high or low and properly handle a return value less than 1 otherwise the player won so we return *False* back to the main loop. If *f_turns_left* is not > *1* we therefore return *False* back to the main loop.

It is VERY important that you experiment by taking out the main *else* block and experiment with the results to see what happens. This will help really solidify these concepts. Taking the time to change values and taking notes of how it effects the final outcome is a good way to

reverse engineer the process and learn. Take the time and experiment with different values and see how it effects the outcome.

We then print the rules.

We then get a random number from *1* to *9* and assign to a *correct_answer* var.

We then create a *guess* var and init to *0*. If we do not do this there is a chance *guess* can be undefined and cause a run-time error.

We then enter our main loop with the condition that *guess != correct_answer* otherwise it will terminate the program. If the *while* loop is true then we assign the return value from *guess_number* into *guess*. If *guess* is not falsy then we enter into the first *if* block and get the return value from *did_win* into a *turns_left* var. If *turns_left* is *False* it will not enter into the other two *if* conditions and re-enter the loop again. If *turns_left == 0* then the game has ended and we break out of the loop. If guess is truthy and *turns_left* is not *0* then we enter into the final *if* block and print the amount of *turns_left* and then re-enter the loop.

There are a number of issues with the above code can you spot them? What happens if you enter in anything other than *1* and *9* does it take guesses away? What happens if you guess the correct number? Does it echo back '*The correct answer is...*'? The question I ask you is should it not take guesses away regardless if a good or bad input and should we tell the player the correct answer if they in fact guess it? What happens if you get the correct answer on the last try? These are things I would like you to think about...

In our first APP we used print statements in our functions. This was deliberate so that we can take small steps together to learn. Going forward we will NOT use any input or print statements in any future functions and/or methods unless we absolutely have to. We will learn how to return str instead and validate information in a more robust way.

Let's properly update our APP.

```
from random import randint


def guess_number(f_guess, f_turns_left):
    """
    Function to obtain player guess

    Params:
```

```
        f_guess: str
        f_turns_left: int

    Returns:
        str, int
    """
    try:
        f_guess = int(f_guess)
        if f_guess < 1 or f_guess > 9:
            raise ValueError
        return f_guess, f_turns_left - 1
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1
    except TypeError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1


def did_win(f_guess, f_correct_answer):
    """
    Function to check player guess against the correct answer

    Params:
        f_guess: int, str
        f_correct_answer: int

    Returns:
        str
    """
    try:
        f_guess = int(f_guess)
        if f_guess > f_correct_answer:
            return 'HINT: Lower Than {0}'.format(f_guess)
        elif f_guess < f_correct_answer:
            return 'HINT: Higher Than {0}'.format(f_guess)
        else:
            return 'You won!'
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.'
    except TypeError:
        return '\nRULES: Please enter a number between 1 and 9.'


print('RULES: Guess a number between 1 and 9.')

correct_answer = randint(1, 9)
turns_left = 3
guess = 0

while guess != correct_answer:
    if turns_left >= 0:
        guess = input('Guess: ')
        guess, turns_left = guess_number(guess, turns_left)
        if guess != correct_answer:
            if turns_left > 1:
                print('{0} turns left.'.format(turns_left))
```

```
            elif turns_left == 1:
                print('{0} turn left.'.format(turns_left))
            else:
                print('The correct answer is {0}, let\'s play
again!'.format(correct_answer))
                break
        game_status = did_win(guess, correct_answer)
        if game_status == 'You won!':
            print(game_status)
            break
        else:
            print(game_status)
```

We see some changes to the logic. We are returning two values in our *guess_number* function which allows us to create two different return values in the main logic.

We also return *str* values and *int* values instead of just raw printing in the functions.

Take some time and really carefully look at each line here to really get a better understanding of proper design. Compare each line with our original and see how the designed has changed.

On final thing is we want to make sure we use exception handling (*try/except*) ONLY in our functions and/or classes where possible. This will handle all input validation in our functions and/or classes and keep that out of our main routine.

This will all come together slowly as we continue to develop.

Take a few hours and work though the above code. Do did you find the logic errors? The program runs but does it perform logically? DO NOT READ AHEAD! Please take a few hours and see if you can sight the logic errors and correct them. Once you think you have them all then continue below.

```
from random import randint


def guess_number(f_guess, f_turns_left):
    """
    Function to obtain player guess

    Params:
        f_guess: str
        f_turns_left: int

    Returns:
        str, int
```

```
    """
    try:
        f_guess = int(f_guess)
        if f_guess < 1 or f_guess > 9:
            raise ValueError
        return f_guess, f_turns_left - 1
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1
    except TypeError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1


def did_win(f_guess, f_correct_answer):
    """
    Function to check player guess against the correct answer

    Params:
        f_guess: int, str
        f_correct_answer: int

    Returns:
        str
    """
    try:
        f_guess = int(f_guess)
        if f_guess > f_correct_answer:
            return 'HINT: Lower Than {0}'.format(f_guess)
        elif f_guess < f_correct_answer:
            return 'HINT: Higher Than {0}'.format(f_guess)
        else:
            return 'You won!'
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.'
    except TypeError:
        return '\nRULES: Please enter a number between 1 and 9.'


print('RULES: Guess a number between 1 and 9.')

correct_answer = randint(1, 9)
turns_left = 3
guess = 0

while guess != correct_answer:
    if turns_left >= 0:
        guess = input('Guess: ')
        guess, turns_left = guess_number(guess, turns_left)
        if guess != correct_answer:
            if turns_left > 1:
                print('{0} turns left.'.format(turns_left))
            elif turns_left == 1:
                print('{0} turn left.'.format(turns_left))
            else:
                print('The correct answer is {0}, let\'s play
again!'.format(correct_answer))
```

```
            break
        game_status = did_win(guess, correct_answer)
        if game_status == 'You won!':
            print(game_status)
            break
        else:
          pass
```

Did you find the following items below in the original code that were better designed above?

- The hint logic in the *did_win* function never gets run.
- The type of the *correct_answer* var is an *int* however the value of the *guess* var is a *str*.

Remove the *f_turns_left - 1* out of the *did_win* function.
The below code should be removed.

```
    if isinstance(guess_response, str):
        print(guess_response)
```

I know this is an overwhelming amount of information. The goal is to experiment with this code so that we can see what happens at each stage. I would recommend you get a piece of paper and write down all of the changes you make and how it effects the outcome of the program when you run it.

TAKE THE TIME AND COMPARE THE THREE VERSIONS OF CODE ABOVE.

Project 5 - Design Wonka Chocolate Machine firmware

It is time to build our biggest project yet!

Mr. Willy Wonka is designing a chocolate vending machine that literally makes the chocolates from a set of raw materials from within the vending machine.

Mr. Willy Wonka is looking for a Firmware Engineer to design the brains of the machine using a microcontroller. He has decided on the micro:bit V2 to control the digital logic of the machine and YOU are now tasked with the role of designing it!

To date most microcontroller firmware has been written in C, C++ and Assembly but now with the advent of MicroPython we have a new option for NEW makers!

Mr. Wonka demands that the machine only take coins so we will have to design that into our firmware. Others have tried to explain that most

people are using credit cards and he screamed, "They shall then have to get change if they want my chocolate!".

His design, in Wonka style, has a tablespoon and teaspoon inside the device that we will directly program to grab the raw materials.

The machine will have 4 chocolate choices which are Dark Chocolate, Mint Chocolate, Caramel Chocolate and Surprise Chocolate.

```
Dark Chocolate @ $2.75
----------------------
1 tablespoon of sugar
1 teaspoon of butter
6 tablespoons of dark chocolate
1 teaspoon of light corn syrup
1 teaspoon of sweetened condensed milk
1 teaspoon of vanilla extract

Mint Chocolate @ $2.50
----------------------
1 tablespoon of sugar
1 teaspoon of butter
6 tablespoons of mint chocolate
1 teaspoon of light corn syrup
1 teaspoon of sweetened condensed milk
1 teaspoon of vanilla extract

Caramel Chocolate @ $3.25
-------------------------
1 tablespoon of sugar
3 tablespoons of caramel
1 teaspoon of butter
6 tablespoons of milk chocolate
1 teaspoon of light corn syrup
1 teaspoon of sweetened condensed milk
1 teaspoon of vanilla extract

Surprise Chocolate @ $3.25
--------------------------
1 tablespoon of sugar
3 tablespoons of Reese's Pieces
1 teaspoon of butter
6 tablespoons of milk chocolate
1 teaspoon of light corn syrup
1 teaspoon of sweetened condensed milk
1 teaspoon of vanilla extract
```

You will be working with the above raw materials when designing your firmware.

Data requirements are as follows.

```
CHOCOLATE_CHOICES = {
    'dark': {
        'ingredients': {
            'sugar': 1,
            'butter': 1,
            'dark chocolate': 6,
            'light corn syrup': 1,
            'sweetened condensed milk': 1,
            'vanilla extract': 1,
        },
        'price': 2.75,
    },
    'mint': {
        'ingredients': {
            'sugar': 1,
            'butter': 1,
            'mint chocolate': 6,
            'light corn syrup': 1,
            'sweetened condensed milk': 1,
            'vanilla extract': 1,
        },
        'price': 2.50,
    },
    'caramel': {
        'ingredients': {
            'sugar': 1,
            'caramel': 3,
            'butter': 1,
            'milk chocolate': 6,
            'light corn syrup': 1,
            'sweetened condensed milk': 1,
            'vanilla extract': 1,
        },
        'price': 3.25,
    },
    'surprise': {
        'ingredients': {
            'sugar': 1,
            'Reese\'s Pieces': 3,
            'butter': 1,
            'milk chocolate': 6,
            'light corn syrup': 1,
            'sweetened condensed milk': 1,
            'vanilla extract': 1,
        },
        'price': 3.25,
    },
}

raw_materials = {
    'sugar': 2,
    'butter': 2,
    'caramel': 15,
    'dark chocolate': 30,
    'mint chocolate': 30,
```

```
    'milk chocolate': 30,
    'light corn syrup': 2,
    'sweetened condensed milk': 2,
    'vanilla extract': 2,
    'Reese\'s Pieces': 15,
}
```

Let's call name it **p_0005_wonka_chocolate_machine.py:**

- Create a **Wonka Chocolate Machine Firmware Requirements** document.
- Create a dictionary called *CHOCOLATE_CHOICES* and populate with the data requirements above.
- Create a *raw_materials* dictionary and populate with the data requirements above.
- Create a *total_money_collected* var and init to *0*.
- Create a *SHUTDOWN_PASSWORD* and init to *'8675309'*.
- Create a *has_raw_materials* function with one param *f_raw_materials*. Create a var *additional_resources_needed* and init to *''*. Create a *for* loop to iterate through the *f_raw_materials* and create conditional logic to check if *f_raw_materials[f_raw_material] > raw_materials[f_raw_material]* and if so create a *additional_resources_needed += 'Machine Needs Additional: {0}\n'.format(f_raw_material)*. Create conditional logic if *additional_resources_needed* then return it else return *True*.
- Create a *collect_money* function with a param *f_max_value*, *f_quarters*, *f_dimes* and *f_nickles* and create a *try/except* block and inside the try block create a *money_collected* var and have *int(f_quaters) * 25* then add to the *money_collected* var and check the *dimes * 0.10* and add to the *money_collected* var and check the *nickels * 0.05*. Create conditional logic to check if ex and if so return a message there are insufficient funds and returning money *elif money_collected >= f_max_value* then return a message the machine can't hold more than *f_max_value* else return *money_collected*. Inside the except block raise a *ValueError* with a message to enter valid currency and return a please enter valid currency message to the main loop.
- Create a *has_enough_money* function with two params *f_money_collected* and *f_chocolate_price*. Create conditional logic to check *f_money_collected >= f_chocolate_price* then within the if block create an *excess_money_collected* var and assign into it *round(f_money_collected - f_chocolate_price, 2)*. Create a global *total_money_collected* var and then *total_money_collected += f_chocolate_price* and then return *'Change: ${0:.2f}\n'.format(excess_money_collected)* back to the main loop *else* print insufficient funds and return an insufficient funds message back to the main loop.

- Create a *bake_chocolate_bar* function with two params *f_chocolate_choice* and *f_raw_materials*. Create a *for* loop and iterate through *f_raw_materials* and *raw_materials[f_raw_material] -= f_raw_materials[f_raw_material]* then outside the for loop return a message that the chocolate bar was dispensed.
- Create a stats function with no params and return each raw material that the machine has and the *total_money_collected*. Start with *cm_stats = 'sugar {0} tablespoons remaining\ n'.format(raw_materials['sugar'])* then += through the rest and return cm_stats to the main program.
- Create a *machine_active* var and init to *True*.
- Create a *choices* list and populate with *'dark'*, *'caramel'*, *'mint'*, *'surprise'*, *'stats'*, *'shutdown'* items.
- Create our main while loop and while *machine_active* create a *valid_choice = False* and get our choice input from the use with the four different choices with their respective prices. Create conditional logic to check if choice in choices then have *valid_choice = True* else print that is not a valid selection. Create another if block to check if *choice == 'shutdown'* and get a password from the user and if it matches what we defined earlier *machine_active = False* which will power down the machine otherwise print a message they are not authorized to disable the machine. Create an *elif choice == 'stats'* call the *stats()* and print the return value and then elif valid_choice and inside *selection = CHOCOLATE_CHOICES[choice]* and create conditional if *has_raw_materials(selection['ingredients'])* and assign that to a var *has_enough_raw_materials* and create conditional logic if not *isinstance(has_enough_raw_materials, bool)* print the variable which will display our *str* from the function and then set *machine_active = False*.
- Create conditional logic for if *isinstance(has_enough_raw_materials, bool)* to then get *quarters = input('Quarters')* same with dimes and nickels. Remember we do not want to cast to an *int* as we do not want an error as the function will handle invalid input. We then run our *collect_money(100.00, quarters, dimes, nickels)* and assign to a var *money*. We then create additional logic if not *isinstance(money, float)* then *print(money) else change = has_enough_money(money, selection['price'])* and additional conditional logic if *change == 'Insufficient funds...'* then p*rint(change) else chocolate_bar = bake_chocolate_bar(choice, selection['ingredients'])* and *print(chocolate_bar)* and *print(change)*. In our larger if block we then create our else then *machine_active = False*.
- Outside of the main loop print we are going down for maintenance.

This is your largest and comprehensive project to date! Take several days and think about each numbered step. Take your time. Experiment and MAKE SURE you thoroughly test out our APP1 so you have a good feel for the project.

If after several hours or days and you get stuck you can find my solution below. Look for the **Part_5_Functions** folder and open the **p_0005_wonka_chocolate_machine folder** in GitHub.

https://github.com/mytechnotalent/Python-For-Kids

I really admire you as you have stuck it out and made it to the end of your fifth step in the MicroPython Journey! Today was insanely challenging! Great job!

In our next lesson we will learn about classes!

Chapter 14: Classes

Today we are going to learn about OOP or object-oriented programming with classes.

This is the lesson of all the lessons thus far where I REALLY want you to take your time! This lesson should be carefully worked between a 2 to 4 week period or until such time that all of the concepts really solidify.

Please take very deliberate care to take breaks and swap out variables and other input data to really get a feel for what is going on at each step of the lesson.

This is an advanced topic and will take patience and endurance. With that said, let's dive in!

By the end of the lesson we will have accomplished the following.

```
* Written a 0014_escape_room where we will build an escape room style adventure game with our micro:bit as we are placed inside a mountain and work our way through different questions inside the cave until we find our way out to the outside world and show our player moving around on our micro:bit display.
```

Classes

When we write larger code to scale it is much easier to take advantage of object-oriented programming or OOP with classes which also make our code less prone to bugs.

In the OOP design structure everything we develop represents real-world items. Let's imagine we were designing software for a car, specifically a Ferrari.

If were were to use what we have learned so far we would use functions to handle all of the functionality of the car so as you can imagine we would literally have thousands of functions to handle everything from handling the engine start to interacting with the brakes and gas handling the navigation software interface, in addition to handling the interaction of literally tens of thousands of individual sensors.

Imagine if you needed to make sweeping changes in the design with functions. Not only would it be very challenging to accomplish as so many different functions rely on each other in a procedural way you are also more prone to making bugs.

If you had say 100 functions that handled the interaction with the steering interface with the engine and the Project Manager and team decided to change out the engine you might possibly have to change every single of those 100 functions!

OOP would be a better solution. The first thing we would do is think about a class that would represent what every car HAS and DOES.

Let's take a step back for a moment. Let's talk about YOU. YOU are a PERSON. Your neighbor is a PERSON. You and your neighbor both may have hair however your neighbor may have brown hair and you have red hair.

If we started off with a generic Person class it would represent all the attributes of what all people share or what each person HAS and what each person DOES.

In classes or OOP you have *attributes* which represent what each person HAS.

In classes or OOP you have *methods* which represent what each person DOES.

```
class Person:
    """
    Base class to represent a generic person
    """

    def __init__(self, name='Generic', has_hair=False, hair_color=None):
        """
        Params:
            name: str, optional
            has_hair: bool, optional
            hair_color: str, optional
        """
        self.name = name
        self.has_hair = has_hair
        self.hair_color = hair_color

    def is_brushing_hair(self):
        """
        Method to handle a person brushing hair if has_hair

        Returns:
            str
        """
        if self.has_hair:
            return '{0} has {1} hair they are brushing.'.format(self.name,
self.hair_color)
        else:
            return '{0} does not have hair.'.format(self.name)
```

```
    def is_eating(self, food):
        """
        Method to handle a person eating

        Params:
            food: str

        Returns:
            str
        """
        return '{0} is eating {1}!'.format(self.name, food)

    def is_walking(self):
        """
        Method to handle a person walking

        Returns:
            str
        """
        return '{0} is walking.'.format(self.name)

    def is_sleeping(self):
        """
        Method to handle a person sleeping

        Returns:
            str
        """
        return '{0} is sleeping.'.format(self.name)
```

Woah what is all that! What is this strange looking *__init__* and *self* stuff all about?

Let's take our time and look at a couple of new and exciting concepts!

When we learned to walk as young children we did not simply enter a marathon on day one of taking our first steps did we?

In fact we had the help of others to stabilize us as we fell down and cried but we GOT BACK UP and tried again.

We are seeing a BRAND NEW design pattern above so let us very carefully take our first step together!

We start out by creating our class called Person and you notice it is not like a function or variable name. It starts with a *capital* letter.

If we were to have multiple words such as *ChocolateMachine* we would create a single class name with each new word capitalized like above.

Earlier we mentioned that classes represent real-world objects. We handle what each object HAS and DOES. We handle an objects *attributes* and m*ethods*. An attribute represents what an object HAS and a method (which is a function that is part of a class) which represents what an object DOES.

As we stabilize our first stand holding desperately on to our parent we begin to become wide-eyed and feel the POWER of this new capability! This is how we will feel when we understand a few of these rather advanced topics. Just as we did when we were little we fell and we fell OFTEN however we KEPT AT IT and today we don't even think about it consciously. This is how you will feel over time about these concepts.

We then see this very strange looking *__init__* method (a method is a function that lives within a class). This is referred to as our constructor. A constructor constructs the very object we are breathing life into! It is where we create our *attributes* or what the object HAS.

The *__init__* is a special method, otherwise referred to a magic method which has special powers in MicroPython.

The very first parameter within the *__init__* method is *self*. The *self* parameter represents the actual what we refer to as *instantiation* or *initialization* of the object we are creating. In our case we could have a *Person* class that we instantiate or create called *katie* for instance.

```
katie = Person('Katie', has_hair=True, hair_color='red')
```

We can take this Class blueprint and immediately make another *Person* called *ted* for instance.

```
ted = Person('Ted')
```

Take note that ted only has one param. If we look back at our constructor or *__init__* method we notice that we have 3 named parameters next to *self*. We have *name='Generic'*, *has_hair=None* and *hair_color=None*.

The *self* parameter in our case is *katie* or *ted*! Let's take this knowledge and now in our mind replace every instance of *self* with *katie* or *ted* and see if this makes more sense.

In our case here, *katie* and *ted* are referred to as an *instance* object and the *Person* class is a *class* object.

To illustrate we will use pseudo code which will NOT run but I want to show you what is going on behind the scenes.

```
def __init__(katie, name='Katie', has_hair=True, hair_color='red'):
    """
    Params:
        name: str. optional
        has_hair: bool, optional
        hair_color: str, optional
    """
    katie.name = name
    katie.has_hair = has_hair
    katie.hair_color = hair_color
```

Let's take this pseudo code which will NOT run but illustrate for our *ted* object.

```
def __init__(ted, name='Ted', has_hair=False, hair_color=None):
    """
    Attrs:
        name: str, optional
        has_hair: bool, optional
        hair_color: str, optional
    """
    ted.name = name
    ted.has_hair = has_hair
    ted.hair_color = hair_color
```

Oh boy we got one foot on the ground! WAY TO GO! A little shaky but with our first foot on the ground gripping on to our parent or guardian we feel a bit of new confidence! We may continue to slip and fall but we will remember this VERY MOMENT and we shall say, 'I CAN TRY AGAIN TILL I GET THIS RIGHT!' It is CRITICAL to understand that the word TILL means WHEN NOT IF!

We now have a *Person* object that HAS *attributes* which are a *name* which is a *str*, *has_hair* which is a *bool* and *hair_color* which is a *NoneType*.

We then see our first non-constructor method which is *is_brushing_hair*. It does not have any params other than our *self* which represents either *katie* or *ted* and we see that it returns a *str*.

We see the following code which contains some conditional logic.

```
if self.has_hair:
    return '{0} has {1} hair they are brushing.'.format(self.name, self.hair_color)
else:
    return '{0} does not have hair.'.format(self.name)
```

Let's make some pseudo code which will NOT run but to help us better understand what is going on.

```
if katie.has_hair:
    return '{0} has {1} hair they are brushing.'.format(katie.name, katie.hair_color)
else:
    return '{0} does not have hair.'.format(katie.name)
```

If *katie*, the instance object, *has_hair* in our instantiation *katie = Person('Katie', has_hair=True, hair_color='red')*, which she does, we will enter into the first if conditional and *return '{0} has {1} hair they are brushing.'.format(self.name, self.hair_color)*. **Keep in mind I put the self back in the return value as we now have a full understanding of what self is.**

The next method, *is_eating*, simply takes a *food* param and returns a formatted *str*.

The *is_walking* method takes no params and simply returns a formatted *str*.

The *is_sleeping* method takes no params and returns a formatted *str*.

Let's create a main routine for our classes.

```
shakira = Person('Shakira', has_hair=True, hair_color='red')
mike = Person('Mike')

brushing_hair = shakira.is_brushing_hair()
print(brushing_hair)

does_not_have_hair = mike.is_brushing_hair()
print(does_not_have_hair)
```

Let's run this code and examine the output.

```
Shakira has red hair they are brushing.
Mike does not have hair.
```

We see that like our *katie* and *ted* examples above a similar outcome. In this case the *shakira* instance object was constructed or initalized with a *'Shakira' name* argument, *has_hair=True* argument and *hair_color='red'* argument. In our *mike* instance object we only pass

in a *name* arg which will cause the *is_brushing_hair* method to handle a different set of conditional logic and give a different return statement.

Let's look at what happens when we construct an bare instance object.

```
generic = Person()

does_not_have_hair = generic.is_brushing_hair()
print(does_not_have_hair)

is_eating_food = generic.is_eating('pizza')
print(is_eating_food)

is_walking_around = generic.is_walking()
print(is_walking_around)

is_sleeping_ = generic.is_sleeping()
print(is_sleeping_)
```

Let's run and see what happens.

```
Generic does not have hair.
Generic is eating pizza!
Generic is walking.
Generic is sleeping.
```

We can see what is going on here. This is where I would like you to take at least an hour and try all of these above examples with different values so you have a good grasp on what is going on.

We have to take careful consideration when naming our variables and make sure that it does not collide with a method or attribute name space. If you noticed we used *is_sleeping_* with an extra _ at the end in order to not collide with the *is_sleeping* method.

Now that we are standing on both feel with our parent or guardian we are going to take our next step by making a custom game engine from scratch using OOP!

Within our micro:bit MicroPython editor there is an ability to create additional files as shown below.

Let us first load up the alpha editor.

https://python.microbit.org/v/alpha

Notices the *Files* folder on the left-hand side of the page.

In the upper left-hand side once you click that icon you will see a little piece of paper with a + sign within it.

All you have to do is create the .py file without using the **.py** extension. For example if you had a file called **player.py** you can simply type **player** and click *Create*.

In the video portion of this course which you will find in the GitHub repo below, Katie will walk you through, step-by-step through this process.

https://github.com/mytechnotalent/Python-For-Kids

Let's name our project **0014_escape_room** within the micro:bit MicroPython Web Editor.

We already have a **main.py** file. Let's add the following files.

config.py
data.py
escape_room_player.py
file_manager.py
game.py
grid.py
player.py

APP 1

Let's create our **Escape Room Application Requirements** document and call it **0014_escape_room_ar**:

```
Escape Room Application Requirements
------------------------------------
1. Define the purpose of the application.
   a. Create a game where we build an escape room style adventure game with
      our micro:bit with questions as we are placed inside a mountain cave
      and work our way through different places in the cave where we randomly
      stumble upon a question space and one of the questions spaces when
      answered correctly gives us a red key which we will need for the other
      randomly placed question.  Once you answer the question which gives you
      the red key you can then answer the question that will let you out of the
      cave or room to the outside world.  The micro:bit display will show an
      outline of the cave and the player moving around a map.
2. Define the rules of the application.
   a. If the player answers a correct question and that question has the red key
      then a red key will be placed into a persistent inventory.
   b. If the player answers the final room question and they have the red key in
      their inventory then they escape to the outside and win.
3. Define the logical steps of the application.
   a. Edit our data.py and populate our database.
```

```
   b. Build out our config.py class logic.
   c. Build out our grid.py class logic.
   d. Build out our player.py base class logic.
   e. Build out our escape_room_player.py child class logic.
   f. Build out our file_manager.py class logic.
   g. Build out our game.py class logic.
   h. Build out our main.py logic to properly account for the integration of all
      of the above classes and guarantee proper functionality.
```

Woah that's a lot to do! The key is we are taking our time. When Leonardo da Vinci painted the Mona Lisa he did not simply snap his fingers and it all appear. Very careful thought and planning took place. We are no different in this matter.

Let us start by populating our **data.py** below.

```
questions = {
    'What year was the MicroBit educational foundation created?':
    [
        '2016',
        '2014',
        '2017',
        0
    ],
    'What year was the first computer invented?':
        [
            '1954',
            '1943',
            '1961',
            1
        ],
    'What year did Damien George create MicroPython?':
        [
            '2015',
            '2012',
            '2014',
            2
        ],
    'What year did the Commodore 64 get released?':
        [
            '1983',
            '1984',
            '1982',
            2
        ],
}
```

Here we have a dictionary with 4 items to which each key has a value that is a list. The first three elements (0 through 2) are the possible correct answers and the fourth element (3) is our correct answer.

Take a moment and review the **Study Buddy** application as it uses this technique in Step 17.

https://github.com/mytechnotalent/MicroPython-micro-bit-Study-Buddy

Now it is time to pour our foundation. When working with OOP we are creating an engine where the base classes can be used over an over in future applications. We are building a game engine as well as an escape room game so the best part about this journey is we will have an engine we can take and create a new game on top of when we are complete which you will do!

We start with a **config.py** file. This nandles any hardware config you see fit.

```
from microbit import pin2

pin2.set_touch_mode(pin2.CAPACITIVE)
```

We then build out the Grid class as we first need to handle all of the environment logic. This class that can be re-used on other MicroPython and even CPython applications!

Let's build out our **grid.py** file and go over each line very carefully!

```
class Grid:
    """
    Class to represent a micro:bit grid
    """

    def __init__(self, led_height=0, led_width=0, led_on='9', led_off='0'):
        """
        Params:
            led_height: int, optional
            led_width: int, optional
            led_on: str, optional
            led_off: str, optional
        """
        self.led_height = led_height
        self.led_width = led_width
        self.led_on = led_on
        self.led_off = led_off
        self.available_height = led_height - 2
        self.available_width = led_width - 2

    def __create(self):
        """
        Private method to create a grid
```

```
        Returns:
            str, str, str
        """
        top_wall = self.led_on * self.led_width + '\n'
        side_walls = ''
        for _ in range(self.available_height):
            side_walls += self.led_on + self.led_off * \
                self.available_width + self.led_on + '\n'
        bottom_wall = self.led_on * self.led_width
        return top_wall, side_walls, bottom_wall

    def update(self, player):
        """
        Method to handle update with each event where we re-draw
        grid with player's current position

        Params:
            player: object

        Returns:
            grid: str
        """
        top_wall, side_walls, bottom_wall = self.__create()
        grid = top_wall + side_walls + bottom_wall + '\n'
        # Convert to a list so that the element can be mutable to add player char
        temp_grid = list(grid)
        # For each step in y, needs to increment by jumps of row width plus the \n
separating rows
        y_adjustment = (player.dy - 1) * (self.led_width + 1)
        # The index position of player marker in the list-formatted grid
        position = self.led_width + 1 + player.dx + y_adjustment
        temp_grid[position] = self.led_on
        grid = ''
        grid = grid.join(temp_grid)
        return grid
```

We start by creating our *__init__* constructor which is what our *Grid* class HAS or otherwise referred to as its *attributes*. The constructor takes four params which are *led_height*, *led_width*, *led_on* and *led_off* to which we create default values of *0* for the *led_height* and *led_width* and a *'9'* for the *led_on* value and *'0'* for our *led_off* value.

```
def __init__(self, led_height=0, led_width=0, led_on='9', led_off='0'):
```

We then assign the param values that will be passed in to *self* which again represents the actual object upon instantiation. We see an *available_height* which is *2* less than the total *led_height* for the grid and the same thing for the width values. The *available_height* and *available_width* are the spaces the player can actually walk.

```
        self.led_height = led_height
        self.led_width = led_width
        self.led_on = led_on
        self.led_off = led_off
        self.available_height = led_height - 2
        self.available_width = led_width - 2
```

We then create a private method called *__create* that takes no params and returns a *top_wall*, *side_walls* and *bottom_wall*.

```
    def __create(self):
        """
        Private method to create a grid

        Returns:
            str, str, str
        """
        top_wall = self.led_on * self.led_width + '\n'
        side_walls = ''
        for _ in range(self.available_height):
            side_walls += self.led_on + self.led_off * \
                self.available_width + self.led_on + '\n'
        bottom_wall = self.led_on * self.led_width
        return top_wall, side_walls, bottom_wall
```

The above will create the *top_wall* and then iterate through a for loop to create the *side_walls* and then create the *bottom_wall*.

We then handle the update method for each player re-draw.

```
    def update(self, player):
        """
        Method to handle update with each event where we re-draw
        grid with player's current position

        Params:
            player: object

        Returns:
            grid: str
        """
        top_wall, side_walls, bottom_wall = self.__create()
        grid = top_wall + side_walls + bottom_wall + '\n'
        # Convert to a list so that the element can be mutable to add player char
        temp_grid = list(grid)
        # For each step in y, needs to increment by jumps of row width plus the \n
separating rows
        y_adjustment = (player.dy - 1) * (self.led_width + 1)
        # The index position of player marker in the list-formatted grid
        position = self.led_width + 1 + player.dx + y_adjustment
        temp_grid[position] = self.led_on
        grid = ''
```

```
        grid = grid.join(temp_grid)
        return grid
```

Now we have taken our first few steps and holding on tight to our parent or guardian and begin to make our way around the room!

We now move on to the **player.py** file with our *Player* base class to create a scalable and reusable Player blueprint for not only this game but for any future games we create!

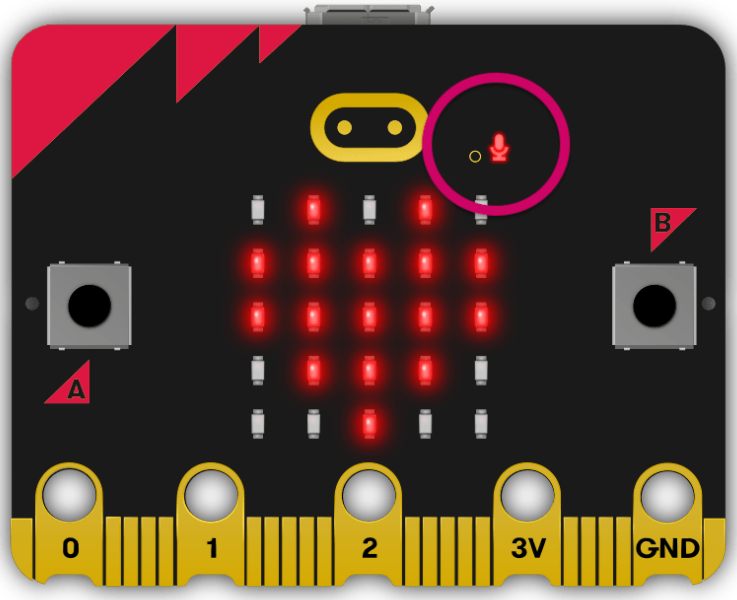
```
from time import sleep


class Player:
    """
    Base class to represent a generic player
    """

    def __init__(self, name, dx, dy, location=None, armour=None, inventory=None):
        """
        Params:
            name: str
            dx: int
            dy: int
            location: tuple, optional
            armour = list, optional
            inventory: list, optional
        """
        self.name = name
        self.dx = dx
        self.dy = dy
        if armour is None:
            armour = []
        self.armour = armour
        if inventory is None:
            inventory = []
        self.inventory = inventory
        self.location = location

    def __move(self, dx, dy):
        """
        Private method to move a generic player based on their current x and y location

        Params:
            dx: int
            dy: int
        """
        self.dx += dx
        self.dy += dy

    @staticmethod
    def get_inventory(file_manager):
        """
```

```
        Static method to get the player inventory from disk

        Params:
            file_manager: object

        Returns:
            str
        """
        inventory = file_manager.read_inventory_file()
        return inventory

    def move_east(self, grid):
        """
        Method to move the player east one position

        Params:
            grid: object
        """
        if self.dx < grid.available_width:
            self.__move(dx=1, dy=0)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)

    def move_west(self, grid):
        """
        Method to move the player east one position

        Params:
            grid: object
        """
        # If the  player is against the left wall do NOT allow them to go through it
        if self.dx != 1 and self.dx <= grid.available_width:
            self.__move(dx=-1, dy=0)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)

    def move_north(self, grid):
        """
        Method to move the player north one position

        Params:
            grid: object
        """
        # If the player is against the top wall do NOT allow them to go through it
        if self.dy != 1 and self.dy <= grid.available_height:
            self.__move(dx=0, dy=-1)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)

    def move_south(self, grid):
        """
        Method to move the player south one position
```

```
        Params:
            grid: object
        """
        if self.dy < grid.available_height:
            self.__move(dx=0, dy=1)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)
```

We start by examining the constructor. We create a *name* param and create a *dx* and *dy* values to represent the player movements along the x and y axis and make them private. The *dx* and *dy* values simply represent the change of position of where the current x and y values are or where are player is on the grid.

Let's illustrate this by looking at an x, y coordinate system on our micro:bit as this is directly from the micro:bit documentation.



Here we see a heart displayed. This is nothing more than 25 LED's in a grid. In the upper-left hand corner we have 0, 0 and in the bottom-right hand corner we have 4, 4. The first number represents the X coordinate and the second number represents the Y coordinate.

The X coordinates run left-right and the Y coordinates run top-bottom.

In our generic base class our player is set to *0, 0* which is *0* on the *x* and *0* on the *y*. The *x* axis goes from left to right and the *y* axis

goes from top to bottom therefore *0, 0* will be the top left led of the 25 total spaces.
To solidify this if we are at *0, 0* we CANNOT move north and we CANNOT move west as we are at the extreme northwest coordinate of our 5 x 5 grid.

Please take some time and read this page of the micro:bit documentation to better understand this very important concept.

https://microbit-micropython.readthedocs.io/en/v2-docs/display.html

As we get back to looking at our constructor we then see an *armour* param which we set to *None* as well as an *inventory* param which we set to *None*.

```
def __init__(self, name, dx, dy, location=None, armour=None, inventory=None):
```

We then assign our params into the *self* values that we have seen before however the *armour* and *inventory* are handled differently. Let's dive into this!

```
        self.name = name
        self.dx = dx
        self.dy = dy
        if armour is None:
            armour = []
        self.armour = armour
        if inventory is None:
            inventory = []
        self.inventory = inventory
        self.location = location
```

Why do we do this? Why can't se simply assign our list [] in the params?

Part of this journey is to use Google and use StackOverflow to answer questions. I want to take this time to take a StackOverflow post which I would like you to read to illustrate why we follow this pattern.

https://stackoverflow.com/questions/4535667/python-list-should-be-empty-on-class-instance-initialisation-but-its-not-why

What we learned is that it is better to set the params to *None* when dealing with a mutable object like a list or dictionary and then assign it inside the body of the constructor to avoid conflicts.

We then create the public methods to expose to main.

```
    def move_east(self, grid):
        """
        Method to move the player east one position

        Params:
            grid: object
        """
        if self.dx < grid.available_width:
            self.__move(dx=1, dy=0)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)

    def move_west(self, grid):
        """
        Method to move the player east one position

        Params:
            grid: object
        """
        # If the  player is against the left wall do NOT allow them to go through it
        if self.dx != 1 and self.dx <= grid.available_width:
            self.__move(dx=-1, dy=0)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)

    def move_north(self, grid):
        """
        Method to move the player north one position

        Params:
            grid: object
        """
        # If the player is against the top wall do NOT allow them to go through it
        if self.dy != 1 and self.dy <= grid.available_height:
            self.__move(dx=0, dy=-1)
        sleep(0.25)
        self.location = self.dx, self.dy
        grid.update(self)

    def move_south(self, grid):
        """
        Method to move the player south one position

        Params:
            grid: object
        """
        if self.dy < grid.available_height:
            self.__move(dx=0, dy=1)
        sleep(0.25)
        self.location = self.dx, self.dy
```

```
        grid.update(self)
```

You can extrapolate the same logic for the other three methods.

Finally we have the *get_inventory* method which works takes the *file_manager* instance object and reads the contents of the *inventory* file and returns it.

We use *@staticmethod* to designate that this method does not handle or access any attributes or methods within the class. When such an situation arises use this decorator as this will be more efficient during runtime.

```
    @staticmethod
    def get_inventory(file_manager):
        """
        Static method to get the player inventory from disk

        Params:
            file_manager: object

        Returns:
            str
        """
        inventory = file_manager.read_inventory_file()
        return inventory
```

Now that we have a robust *Player* base class we can create an *escape_room_player* subclass or child class within our **escape_room_player.py** file so let's build it out.

```
from player import Player


class EscapeRoomPlayer(Player):
    """
    Child class to represent an escape room player
    """

    def __init__(self, name, dx=1, dy=1, location=None):
        """
        Parms:
            name: str
            dx: int, optional
            dy: int, optional
            location: tuple, optional
        """
        super().__init__(name, dx, dy, location)

    @staticmethod
    def pick_up_red_key(game):
```

```
        """
        Static method to handle picking up red key

        Params:
            game: object

        Returns:
            str
        """
        game.write_inventory_file('Red Key')
        return 'You picked up the red key!'

    @staticmethod
    def without_red_key():
        """
        Static method to handle not having the red key

        Returns:
            str
        """
        return 'You do not have the red key to escape.'
```

We start by calling the *super()* on the parent constructor. We override and set *dx* and *dy* in our params to *1* as that is where the beginning bounds of our player movement will be and we will therefore set the player on init or power on to *1, 1*. Calling *super().__init__* and then passing in the attributes allows us to inherit all of those features or what it HAS for free from the parent class.

```
    def __init__(self, name, dx=1, dy=1, location=None):
        """
        Parms:
            name: str
            dx: int, optional
            dy: int, optional
            location: tuple, optional
        """
        super().__init__(name, dx, dy, location)
```

We then have the *pick_up_red_key* method to write the *str* or *'Red Key'* into our inventory file and return the value.

```
    @staticmethod
    def pick_up_red_key(game):
        """
        Static method to handle picking up red key

        Params:
            game: object

        Returns:
            str
```

```
        """
        game.write_inventory_file('Red Key')
        return 'You picked up the red key!'
```

We then handle the *without_red_key* logic if the player does not have the key which simply returns a *str*.

```
    def without_red_key(self):
        """
        Method to handle not having the red key

        Returns:
            str
        """
        return 'You do not have the red key to escape.'
```

We then build out our **file_manager.py** file which will handle writing our inventory values into the micro:bit flash storage.

```
class FileManager:
    """
    Class to implement file access to store inventory to maintain persistence
    """

    def __init__(self):
        self.inventory = ''

    @staticmethod
    def clear_inventory_file():
        """
        Static method to clear inventory file after winning a game

        Returns:
            bool
        """
        try:
            with open('inventory', 'w') as file:
                file.write('')
                return True
        except OSError:
            return False

    @staticmethod
    def write_inventory_file(inventory_item):
        """
        Static method to write inventory item to inventory file upon picking it up

        Params:
            inventory_item: str

        Returns:
            bool
```

```
        """
        try:
            with open('inventory', 'w') as file:
                file.write(inventory_item)
                return True
        except OSError:
            return False

    def read_inventory_file(self):
        """
        Method to read inventory file and return its contents

        Return:
            str or bool
        """
        try:
            with open('inventory', 'r') as file:
                self.inventory = file.read()
                return self.inventory
        except OSError:
            return False
```

We have three static methods with no constructor. Let us take a look at how to manage simple file access.

```
        try:
            with open('inventory', 'r') as file:
                inventory = file.read()
                return inventory
        except OSError:
            return False
```

Inside our *try/except* block we make sure that handle the fact that the file does not exist yet and to simply ignore it. We then use a *with* keyword which is a context manager. What that means is all the instructions that are within the with block are local to that with block. This will allow you to properly close a file without you having to type additional code.

```
            with open('inventory', 'r') as file:
```

The above will attempt to read a file called 'inventory', which at this point does not exist. If it did it will read the file in and create a file object called file.

```
                inventory = file.read()
```

Here we read all the contents of the file and store the *str* into an *inventory* var and simply return it.

```
    @staticmethod
    def write_inventory_file(self, inventory_item):
        """
        Static method to write inventory item to inventory file upon picking it up

        Returns:
            bool
        """
        try:
            with open('inventory', 'w') as file:
                file.write(inventory_item)
                return True
        except OSError:
            return False
```

Here we pass in an *inventory_item* param and literally open a file called inventory and if it does not exist create it.

```
            with open('inventory', 'w') as file:
```

Then we take the file object and write the value of the param *inventory_file* into it and close the file. Therefore when the inventory value is passed in, *'Red Key'*, a *str*, will be written to a file called **inventory**.

The final method simply overwrites the inventory file and puts in an empty string.

```
    @staticmethod
    def clear_inventory_file():
        """
        Static method to clear inventory file after winning a game

        Returns:
            bool
        """
        try:
            with open('inventory', 'w') as file:
                file.write('')
                return True
        except OSError:
            return False
```

Let's build out the **game.py** file for our question logic.

First we are going to work with our **data.py** so let's review that code first before diving into the *Game* glass to make sure we understand data structures!

```
questions = {
    'What year was the MicroBit educational foundation created?':
    [
        '2016',
        '2014',
        '2017',
        0
    ],
    'What year was the first computer invented?':
        [
            '1954',
            '1943',
            '1961',
            1
        ],
    'What year did Damien George create MicroPiethon?':
        [
            '2015',
            '2012',
            '2014',
            2
        ],
    'What year did the Commodore 64 get released?':
        [
            '1983',
            '1984',
            '1982',
            2
        ],
}
```

Now let's examine our class below.

```
from random import randint, choice
from microbit import display, Image
from speech import say
import music

from data import questions
from file_manager import FileManager


class Game:
    """
    Class to handle game integration
    """

    def __init__(self):
        self.SAY_SPEED = 95
        self.final_question = False
        self.random_question = None
        self.answer_1 = None
        self.answer_2 = None
        self.answer_3 = None
```

```
        self.correct_answer_index = None
        self.correct_answer = None
        self.file_manager = FileManager()
        self.__instructions()

    @staticmethod
    def __generate_random_number(grid):
        """
        Private static method to handle obtaining random number

        Params:
            grid: object

        Returns:
            int
        """
        x = randint(1, grid.available_width)
        return x

    @staticmethod
    def __generate_random_numbers(grid):
        """
        Private static method to handle obtaining random numbers

        Params:
            grid: object

        Returns:
            int, int
        """
        x = randint(1, grid.available_width)
        y = randint(1, grid.available_height)
        while x == 1 and y == 1:
            x = randint(1, grid.available_width)
            y = randint(1, grid.available_width)
        return x, y

    @staticmethod
    def __correct_answer_response():
        """
        Private static method to handle correct answer response

        Returns:
            str
        """
        return '\nCorrect!'

    def __instructions(self):
        """
        Private method to give instructions to the player
        """
        display.show(Image.SURPRISED)
        say('Welcome to the Escape Room!', speed=self.SAY_SPEED)
        say('Press the aay button to move west.', speed=self.SAY_SPEED)
        say('Press the lowgo to move north.', speed=self.SAY_SPEED)
```

```
        say('Press the bee button to move east.', speed=self.SAY_SPEED)
        say('Press pin two to move south.', speed=self.SAY_SPEED)
        say('Good luck!', speed=self.SAY_SPEED)
        say('Let the games begin!', speed=self.SAY_SPEED)

    def __ask_random_question(self, d_questions):
        """
        Private method to ask a random question from the database

        Params:
            d_questions: dict
        """
        self.random_question = choice(list(d_questions))
        self.answer_1 = d_questions[self.random_question][0]
        self.answer_2 = d_questions[self.random_question][1]
        self.answer_3 = d_questions[self.random_question][2]
        self.correct_answer_index = d_questions[self.random_question][3]
        self.correct_answer = d_questions[self.random_question]
[self.correct_answer_index]

    def __incorrect_answer_response(self):
        """
        Private method to handle incorrect answer logic

        Params:
            correct_answer: str

        Returns:
            str
        """
        return '\nIncorret.  The correct answer is {0}.'.format(self.correct_answer)

    def __win(self):
        """
        Private method to handle win game logic

        Returns:
            str
        """
        self.file_manager.clear_inventory_file()
        return '\nYou Escaped!'

    def generate_question(self, grid, player):
        """
        Method to generate a question

        Params:
            grid: object
            player: object

        Returns:
            bool
        """
        self.__ask_random_question(questions)
        random_location = (x, y) = self.__generate_random_numbers(grid)
```

```
        if self.random_question and random_location == player.location:
            display.show(Image.SURPRISED)
            say('You found gold!', speed=self.SAY_SPEED)
            say('Answer the question correctly.', speed=self.SAY_SPEED)
            say(self.random_question, speed=self.SAY_SPEED)
            say('Press the aay button for {0}.'.format(self.answer_1),
speed=self.SAY_SPEED)
            say('Press the lowgo for {0}.'.format(self.answer_2), speed=self.SAY_SPEED)
            say('Press the bee button for {0}.'.format(self.answer_3),
speed=self.SAY_SPEED)
            display.show(Image.HAPPY)
            return True
        else:
            return False

    def did_player_win(self, grid, player, player_response):
        """
        Method to handle the check if player won

        Params:
            grid: object
            player: object
            player_response: int

        Returns:
            bool
        """
        if isinstance(self.correct_answer_index, int):
            if player_response == self.correct_answer_index + 1:
                display.show(Image.SURPRISED)
                say(self.__correct_answer_response(), speed=self.SAY_SPEED)
                inventory = player.get_inventory(self.file_manager)
                player.inventory.append(inventory)
                if 'Red Key' in player.inventory:
                    display.show(Image.SURPRISED)
                    say(self.__win(), speed=self.SAY_SPEED)
                    music.play(music.POWER_UP)
                    display.show(Image.ALL_CLOCKS, loop=False, delay=100)
                    return True
                elif 'Red Key' not in player.inventory and not self.final_question:
                    receive_red_key = self.__generate_random_number(grid)
                    if receive_red_key == 2:
                        display.show(Image.SURPRISED)
                        say(player.pick_up_red_key(self.file_manager),
speed=self.SAY_SPEED)
                        self.final_question = True
                        return False
                    else:
                        display.show(Image.SURPRISED)
                        say(player.without_red_key(), speed=self.SAY_SPEED)
                        return False
            else:
                display.show(Image.SURPRISED)
                say(self.__incorrect_answer_response(), speed=self.SAY_SPEED)
                return False
```

The first thing we do is import our required imports.

```
from random import randint, choice
from microbit import display, Image
from speech import Say
import music

from data import questions
from file_manager import FileManager
```

The *__generate_random_number* and *__generate_random_numbers* provide private methods to get a random number for key placement and a random series of numbers for question placement.

```
    @staticmethod
    def __generate_random_number(grid):
        """
        Private static method to handle obtaining random number

        Params:
            grid: object

        Returns:
            int
        """
        x = randint(1, grid.available_width)
        return x

    @staticmethod
    def __generate_random_numbers(grid):
        """
        Private static method to handle obtaining random numbers

        Params:
            grid: object

        Returns:
            int, int
        """
        x = randint(1, grid.available_width)
        y = randint(1, grid.available_height)
        while x == 1 and y == 1:
            x = randint(1, grid.available_width)
            y = randint(1, grid.available_width)
        return x, y
```

We then have an *ask_random_question* method that will ask a random question from our dictionary which is a database.

```
    def __ask_random_question(self, d_questions):
        """
        Private method to ask a random question from the database

        Params:
            d_questions: dict
        """
        self.random_question = choice(list(d_questions))
        self.answer_1 = d_questions[self.random_question][0]
        self.answer_2 = d_questions[self.random_question][1]
        self.answer_3 = d_questions[self.random_question][2]
        self.correct_answer_index = d_questions[self.random_question][3]
        self.correct_answer = d_questions[self.random_question]
[self.correct_answer_index]
```

Let's take this opportunity to learn how to do some print debugging. This code will not run for you yet as you have not built out the engine however for the purposes of learning I will show you directly what *random_question* produces.

```
        self.random_question = choice(list(d_questions))
        print(random_question)
```

When we run this code and step on a space that has a question we get *random_question* printed which contains the value of a random question.

```
>>> What year did the Commodore 64 get released?
```

We then see *answer_1*, *answer_2* and *answer_3* so let's debug *answer_1* and see what is inside *answer_1*.

```
>>> 1983
```

You can see how we can work with data as we see the above result.

The next thing we are going to look at is the *correct_answer_index* and *correct_answer* vars.

```
        self.correct_answer_index = questions[self.random_question][3]
        self.correct_answer = questions[self.random_question][self.correct_answer_index]
```

Let's look at our **data.py** specific question.

```
    'What year did the Commodore 64 get released?':
        [
            '1983',
            '1984',
```

```
            '1982',
            2
        ],
```

Our *correct_answer_index* will hold the literal integer *2* and our *correct_answer* will hold the value at index *2* or *'1982'* the *str*.

THE MOST IMPORTANT CONCEPT IN SOFTWARE ENGINEERING IS MASTERING CONDITIONAL LOGIC!

THE SECOND MOST IMPORTANT CONCEPT IN SOFTWARE ENGINEERING IS MASTERING DATA!

Data is everywhere. Data is the new oil. Let's take a moment and really properly understand how dictionaries hold data by taking a second look above.

```
    'What year did the Commodore 64 get released?':
        [
            '1983',
            '1984',
            '1982',
            2
        ],
```

We have a *key* which is the question and a *list* of values of which 3 are *str* and the 4th is an *int*.

```
questions[random_question][3]
```

Here we have the *questions* database in our case or more simply the *questions* dictionary.

```
questions = {
    'What year was the MicroBit educational foundation created?':
    [
        '2016',
        '2014',
        '2017',
        0
    ],
    'What year was the first computer invented?':
        [
            '1954',
            '1943',
            '1961',
            1
        ],
    'What year did Damien George create MicroPiethon?':
        [
```

```
            '2015',
            '2012',
            '2014',
            2
        ],
    'What year did the Commodore 64 get released?':
        [
            '1983',
            '1984',
            '1982',
            2
        ],
}
```

Therefore…

```
questions[self.random_question][3]
```

In this above, questions is the *dictionary* and we are reaching inside and getting the *random_question* value.

```
self.random_question = choice(list(questions))
```

This is how we get a *random_question*. We know that *choice()* is going to grab a random value.

```
list(questions)
```

This will get a single key from a dictionary. Therefore an example of a single key is the following.

```
'What year did Damien George create MicroPiethon?'
```

Let's re-examine the above statement and substitute in the value.

```
questions[self.random_question][3]
```

Then becomes the following.

```
questions['What year did Damien George create MicroPiethon?'][3]
```

Let's look at this specific key/value pair.

```
    'What year did Damien George create MicroPiethon?':
        [
            '2015',
            '2012',
```

```
            '2014',
            2
        ],
```

We see if we take the *questions* dictionary and get the *'What year did Damien George create MicroPython?'*, it will return a list. REMEMBER we misspell MicroPython deliberately so the voice module can better speak it.

```
questions['What year did Damien George create MicroPiethon?']
```

That alone will return the following.

```
        [
            '2015',
            '2012',
            '2014',
            2
        ],
```

We see if we then examine the following.

```
questions['What year did Damien George create MicroPiethon?'][3]
```

This will give us the fourth element, as our lists are 0 indexed, in the list or *[3]* which is *2* which is an *int*.

```
            2
```

Therefore if you grab the following.

```
questions[self.random_question][self.correct_answer_index]
```

It will give you *'2014'*.

Mini data lesson complete! Finally, back to our method! We return all these strings.

The final methods all handle the responses which are pretty self explanatory.

```
    @staticmethod
    def __correct_answer_response(self):
        """
        Private static method to handle correct answer response

        Returns:
```

```
            str
        """
        return '\nCorrect!'

    def __incorrect_answer_response(self):
        """
        Private method to handle incorrect answer logic

        Params:
            correct_answer: str

        Returns:
            str
        """
        return '\nIncorret.  The correct answer is {0}.'.format(self.correct_answer)

    def __win(self):
        """
        Private method to handle win game logic

        Returns:
            str
        """
        self.file_manager.clear_inventory_file()
        return '\nYou Escaped!'

    def generate_question(self, grid, player):
        """
        Method to generate a question

        Params:
            grid: object
            player: object

        Returns:
            bool
        """
        self.__ask_random_question(questions)
        random_location = (x, y) = self.__generate_random_numbers(grid)
        if self.random_question and random_location == player.location:
            display.show(Image.SURPRISED)
            say('You found gold!', speed=self.SAY_SPEED)
            say('Answer the question correctly.', speed=self.SAY_SPEED)
            say(self.random_question, speed=self.SAY_SPEED)
            say('Press the aay button for {0}.'.format(self.answer_1),
speed=self.SAY_SPEED)
            say('Press the lowgo for {0}.'.format(self.answer_2), speed=self.SAY_SPEED)
            say('Press the bee button for {0}.'.format(self.answer_3),
speed=self.SAY_SPEED)
            display.show(Image.HAPPY)
            return True
        else:
            return False

    def did_player_win(self, grid, player, player_response):
```

```
        """
        Method to handle the check if player won

        Params:
            grid: object
            player: object
            player_response: int

        Returns:
            bool
        """
        if isinstance(self.correct_answer_index, int):
            if player_response == self.correct_answer_index + 1:
                display.show(Image.SURPRISED)
                say(self.__correct_answer_response(), speed=self.SAY_SPEED)
                inventory = player.get_inventory(self.file_manager)
                player.inventory.append(inventory)
                if 'Red Key' in player.inventory:
                    display.show(Image.SURPRISED)
                    say(self.__win(), speed=self.SAY_SPEED)
                    music.play(music.POWER_UP)
                    display.show(Image.ALL_CLOCKS, loop=False, delay=100)
                    return True
                elif 'Red Key' not in player.inventory and not self.final_question:
                    receive_red_key = self.__generate_random_number(grid)
                    if receive_red_key == 2:
                        display.show(Image.SURPRISED)
                        say(player.pick_up_red_key(self.file_manager),
speed=self.SAY_SPEED)
                        self.final_question = True
                        return False
                    else:
                        display.show(Image.SURPRISED)
                        say(player.without_red_key(), speed=self.SAY_SPEED)
                        return False
            else:
                display.show(Image.SURPRISED)
                say(self.__incorrect_answer_response(), speed=self.SAY_SPEED)
                return False
```

Finally we are ready to build our app! Let's look at **main.py** and break it down.

```
from microbit import display, Image, button_a, button_b, pin_logo, pin2

from grid import Grid
from game import Game
from escape_room_player import EscapeRoomPlayer

from config import *

grid = Grid(5, 5)
player = EscapeRoomPlayer('Mr. George')
```

```
game = Game()

game_on = True
while game_on:
    display.show(Image(grid.update(player)))

    player_move = True
    while player_move:
        if button_a.is_pressed():
            player.move_west(grid)
            player_move = False
        elif button_b.is_pressed():
            player.move_east(grid)
            player_move = False
        elif pin_logo.is_touched():
            player.move_north(grid)
            player_move = False
        elif pin2.is_touched():
            player.move_south(grid)
            player_move = False

    player_found_gold = game.generate_question(grid, player)
    player_response = None
    while player_found_gold:
        if button_a.is_pressed():
            player_response = 1
            did_player_win = game.did_player_win(grid, player, player_response)
            player_found_gold = False
            if did_player_win:
                game_on = False
        elif pin_logo.is_touched():
            player_response = 2
            did_player_win = game.did_player_win(grid, player, player_response)
            player_found_gold = False
            if did_player_win:
                game_on = False
        elif button_b.is_pressed():
            player_response = 3
            did_player_win = game.did_player_win(grid, player, player_response)
            player_found_gold = False
            if did_player_win:
                game_on = False
```

Ok... this is a good deal of logic.

We first import everything we need.

We then instantiate our instance methods.

- The logic here is very simple to read. What is very important especially when it comes to scalable and testable code is that you try to keep your inputs within your **main.py** file.

- You in addition want to use boolean logic to control your various stages of the game and handle them properly.

Project 6a - Design Your Own OOP Game!

This is part 1 of your capstone project for the course. You will simply take the engine that you built today and will now use the classes you wish to build your OWN game! Take several hours or days and really pour over what you learned here before you begin.

Call it **0015_BLANK_game** and substitute the BLANK for the name of your game as that will be your folder name. Use the naming conventions that you have learned for your files.

Project 6b - Design Wonka Chocolate Machine firmware (OOP)

Take your functional design from the last lesson and start from scratch and make an OOP version!

Call it **p_0006_wonka_chocolate_machine** as that will be your folder name. Use the naming conventions that you have learned for your files.

You can find today as well as all the source code in the GitHub repo if you run into any issues.

https://github.com/mytechnotalent/Python-For-Kids

I really admire you as you have stuck it out and made it to the end of your sixth step in the MicroPython Journey! Today was long and likely took two to four weeks to complete. Great job!

In our next lesson we will cover Unittest!

Chapter 15: Unittest

Today is a most exciting day where we pull together this amazing journey and discuss the concept of Unittest.

Unittest is a way to check our individual Python functions and methods to ensure they are doing what we THINK they are doing with what they are ACTUALLY doing.

The biggest issue a Developer faces is the fact that it is impossible to keep straight all of the different expectations of a function or method in their head. We do not have to with Unittest.

Let’s create a new project within the web editor and upload the MicroPython **unittest.py** module to it.

Let’s copy the contents into our project’s **unittest.py** file.

```
import sys


class SkipTest(Exception):
    """
    Class to handle skipping test functionality
    """
    pass


class AssertRaisesContext:
    """
    Class to handle an assertion raising context
    """

    def __init__(self, exc):
        """
        Params:
            exc: str
        """
        self.expected = exc

    def __enter__(self):
        """
        Magic method to handle enter implementation objects used with the with statement

        Returns:
            str
        """
        return self

    def __exit__(self, exc_type):
        """
```

```
        Magic method to handle exit implementation objects used with the with statement

        Params:
            exc_type: str

        Returns:
            bool
        """
        if exc_type is None:
            assert False, '%r not raised' % self.expected
        if issubclass(exc_type, self.expected):
            return True
        return False


class TestCase:
    """
    Class to handle unittest test case functionality
    """

    def fail(self, msg=''):
        """
        Method to handle fail logic

        Params:
            msg: str, optional
        """
        assert False, msg

    def assertEqual(self, x, y, msg=''):
        """
        Method to handle assert equal logic

        Params:
            x: ANY
            y: ANY
            msg: str, optional
        """
        if not msg:
            msg = '%r vs (expected) %r' % (x, y)
        assert x == y, msg

    def assertNotEqual(self, x, y, msg=''):
        """
        Method to handle assert not equal logic

        Params:
            x: ANY
            y: ANY
            msg: str, optional
        """
        if not msg:
            msg = '%r not expected to be equal %r' % (x, y)
        assert x != y, msg
```

```
    def assertAlmostEqual(self, x, y, places=None, msg='', delta=None):
        """
        Method to handle assert almost equal logic

        Params:
            x: ANY
            y: ANY
            places: NoneType, optional
            msg: str, optional
            delta: NoneType, optional
        """
        if x == y:
            return
        if delta is not None and places is not None:
            raise TypeError('specify delta or places not both')
        if delta is not None:
            if abs(x - y) <= delta:
                return
            if not msg:
                msg = '%r != %r within %r delta' % (x, y, delta)
        else:
            if places is None:
                places = 7
            if round(abs(y-x), places) == 0:
                return
            if not msg:
                msg = '%r != %r within %r places' % (x, y, places)
        assert False, msg

    def assertNotAlmostEqual(self, x, y, places=None, msg='', delta=None):
        """
        Method to handle assert not almost equal logic

        Params:
            x: ANY
            y: ANY
            places: NoneType, optional
            msg: str, optional
            delta: NoneType, optional
        """
        if delta is not None and places is not None:
            raise TypeError("specify delta or places not both")
        if delta is not None:
            if not (x == y) and abs(x - y) > delta:
                return
            if not msg:
                msg = '%r == %r within %r delta' % (x, y, delta)
        else:
            if places is None:
                places = 7
            if not (x == y) and round(abs(y-x), places) != 0:
                return
            if not msg:
                msg = '%r == %r within %r places' % (x, y, places)
        assert False, msg
```

```
def assertIs(self, x, y, msg=''):
    """
    Method to handle assert is logic

    Params:
        x: ANY
        y: ANY
        msg: str, optional
    """
    if not msg:
        msg = '%r is not %r' % (x, y)
    assert x is y, msg

def assertIsNot(self, x, y, msg=''):
    """
    Method to handle assert is not logic

    Params:
        x: ANY
        y: ANY
        msg: str, optional
    """
    if not msg:
        msg = '%r is %r' % (x, y)
    assert x is not y, msg

def assertIsNone(self, x, msg=''):
    """
    Method to handle assert is none logic

    Params:
        x: ANY
        msg: str, optional
    """
    if not msg:
        msg = '%r is not None' % x
    assert x is None, msg

def assertIsNotNone(self, x, msg=''):
    """
    Method to handle assert is not none logic

    Params:
        x: ANY
        msg: str, optional
    """
    if not msg:
        msg = '%r is None' % x
    assert x is not None, msg

def assertTrue(self, x, msg=''):
    """
    Method to handle assert true logic
```

```
        Params:
            x: ANY
            msg: str, optional
        """
        if not msg:
            msg = 'Expected %r to be True' % x
        assert x, msg

    def assertFalse(self, x, msg=''):
        """
        Method to handle assert false logic

        Params:
            x: ANY
            msg: str, optional
        """
        if not msg:
            msg = 'Expected %r to be False' % x
        assert not x, msg

    def assertIn(self, x, y, msg=''):
        """
        Method to handle assert in logic

        Params:
            x: ANY
            y: ANY
            msg: str, optional
        """
        if not msg:
            msg = 'Expected %r to be in %r' % (x, y)
        assert x in y, msg

    def assertIsInstance(self, x, y, msg=''):
        """
        Method to handle assert is instance logic

        Params:
            x: ANY
            y: ANY
            msg: str, optional
        """
        assert isinstance(x, y), msg

    def assertRaises(self, exc, func=None, *args, **kwargs):
        """
        Method to handle assert is instance logic

        Params:
            exc: str
            func: NoneType, optional
            *args: ANY, optional
            **kwargs: ANY, optional
        """
        if func is None:
```

```
            return AssertRaisesContext(exc)
        try:
            func(*args, **kwargs)
            assert False, "%r not raised" % exc
        except Exception as e:
            if isinstance(e, exc):
                return
            raise


def skip(msg):
    """
    Function to handle skip logic

    Params:
        msg: str

    Returns:
        object
    """
    def _decor(fun):
        """
        Inner function to handle private _decor logic

        Params:
            msg: str

        Returns:
            object
        """
        def _inner(self):
            """
            Inner function to handle the replace original fun with _inner

            Params:
                msg: str

            Returns:
                object
            """
            raise SkipTest(msg)
        return _inner
    return _decor


def skipIf(cond, msg):
    """
    Function to handle skip if logic

    Params:
        cond: str
        msg: str

    Returns:
        object
```

```
    """
    if not cond:
        return lambda x: x
    return skip(msg)


def skipUnless(cond, msg):
    """
    Function to handle skip unless logic

    Params:
        cond: str
        msg: str

    Returns:
        object
    """
    if cond:
        return lambda x: x
    return skip(msg)


class TestSuite:
    """
    Class to handle unittest test suite functionality
    """

    def __init__(self):
        self.tests = []

    def addTest(self, cls):
        """
        Method to handle adding a test functionality

        Params:
            cls: str
        """
        self.tests.append(cls)


class TestRunner:
    """
    Class to handle test runner functionality
    """

    def run(self, suite):
        """
        Method to handle test run functionality

        Params:
            suite: object

        Returns:
            object
        """
```

```
        res = TestResult()
        for c in suite.tests:
            run_class(c, res)
        print("Ran %d tests\n" % res.testsRun)
        if res.failuresNum > 0 or res.errorsNum > 0:
            print("FAILED (failures=%d, errors=%d)" % (res.failuresNum, res.errorsNum))
        else:
            msg = "OK"
            if res.skippedNum > 0:
                msg += " (%d skipped)" % res.skippedNum
            print(msg)
        return res


class TestResult:
    """
    Class to handle test result functionality
    """

    def __init__(self):
        self.errorsNum = 0
        self.failuresNum = 0
        self.skippedNum = 0
        self.testsRun = 0

    def wasSuccessful(self):
        """
        Method to handle indication of a successful test functionality

        Returns:
            bool
        """
        return self.errorsNum == 0 and self.failuresNum == 0


def run_class(c, test_result):
    """
    Function to handle running of class functionality

    Params:
        c: object
        test_result: bool
    """
    o = c()
    set_up = getattr(o, 'setUp', lambda: None)
    tear_down = getattr(o, 'tearDown', lambda: None)
    for name in dir(o):
        if name.startswith('test'):
            print('%s (%s) ...' % (name, c.__qualname__), end='')
            m = getattr(o, name)
            set_up()
            try:
                test_result.testsRun += 1
                m()
                print(' ok')
```

```
            except SkipTest as e:
                print(' skipped:', e.args[0])
                test_result.skippedNum += 1
            except:
                print(' FAIL')
                test_result.failuresNum += 1
                raise
            finally:
                tear_down()


def main(module='__main__'):
    def test_cases(m):
        """
        Function to handle test case running functionality

        Params:
            m: object
        """
        for tn in dir(m):
            c = getattr(m, tn)
            if isinstance(c, object) and isinstance(c, type) and issubclass(c, TestCase):
                yield c


    m = __import__(module)
    suite = TestSuite()
    for c in test_cases(m):
        suite.addTest(c)
    runner = TestRunner()
    result = runner.run(suite)
    # Terminate with non zero return code in case of failures
    sys.exit(result.failuresNum > 0)
```

Let's revisit our **0013_number_guessing_game_repl** app except this time let's break out the functions into their own module called **funcs.py** like below.

```
def guess_number(f_guess, f_turns_left):
    """
    Function to obtain player guess

    Params:
        f_guess: str
        f_turns_left: int

    Returns:
        int, int or str, int
    """
    try:
        f_guess = int(f_guess)
        if f_guess < 1 or f_guess > 9:
            raise ValueError
```

```
        return f_guess, f_turns_left - 1
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.', f_turns_left - 1


def did_win(f_guess, f_correct_answer):
    """
    Function to check player guess against the correct answer

    Params:
        f_guess: int or str
        f_correct_answer: int

    Returns:
        str
    """
    try:
        f_guess = int(f_guess)
        if f_guess > f_correct_answer:
            return 'HINT: Lower Than {0}'.format(f_guess)
        elif f_guess < f_correct_answer:
            return 'HINT: Higher Than {0}'.format(f_guess)
        else:
            return 'You won!'
    except ValueError:
        return '\nRULES: Please enter a number between 1 and 9.'
```

As we can see we have a *guess_number* and *did_win* function which we broke out from **main.py**. Let's create a file called **test_funcs.py** and add the following.

```
import unittest

from funcs import guess_number, did_win


class TestFuncs(unittest.TestCase):
    """
    Test class to test funcs module
    """

    def test_guess_number(self):
        """
        test guess_number functionality
        """
        # Params
        f_guess = '4'
        f_turns_left = 3
        # Returns
        return_1 = 4
        return_2 = 2
        # Calls
        integer_1, integer_2 = guess_number(f_guess, f_turns_left)
```

```
        # Asserts
        self.assertEqual(integer_1, return_1)
        self.assertEqual(integer_2, return_2)

    def test_guess_number_incorrect_integer_range(self):
        """
        test guess_number functionality handles incorrect integer range
        """
        # Params
        f_guess = '-5'
        f_turns_left = 3
        # Returns
        return_1 = '\nRULES: Please enter a number between 1 and 9.'
        return_2 = 2
        # Calls
        string_1, integer_1 = guess_number(f_guess, f_turns_left)
        # Asserts
        self.assertEqual(string_1, return_1)
        self.assertEqual(integer_1, return_2)

    def test_guess_number_non_integer(self):
        """
        test guess_number functionality handles non integer
        """
        # Params
        f_guess = 'k'
        f_turns_left = 3
        # Returns
        return_1 = '\nRULES: Please enter a number between 1 and 9.'
        return_2 = 2
        # Calls
        string_1, integer_1 = guess_number(f_guess, f_turns_left)
        # Asserts
        self.assertEqual(string_1, return_1)
        self.assertEqual(integer_1, return_2)

    def test_did_win(self):
        """
        test did_win functionality
        """
        # Params
        f_guess = 5
        f_correct_answer = 5
        # Returns
        return_1 = 'You won!'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)

    def test_did_win_guessed_lower(self):
        """
        test did_win functionality handles guessed lower
        """
        # Params
```

```
        f_guess = 4
        f_correct_answer = 5
        # Returns
        return_1 = 'HINT: Higher Than 4'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)

    def test_did_win_guessed_higher(self):
        """
        test did_win functionality handles guessed higher
        """
        # Params
        f_guess = 6
        f_correct_answer = 5
        # Returns
        return_1 = 'HINT: Lower Than 6'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)

    def test_did_win_non_integer(self):
        """
        test did_win functionality handles non integer
        """
        # Params
        f_guess = 'k'
        f_correct_answer = 5
        # Returns
        return_1 = '\nRULES: Please enter a number between 1 and 9.'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)


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

Let's break this down. We first import our two functions from the **funcs** module and then create a class called *TestFuncs* and have it as a child class to *unittest.TestCase*.

We then create our first test to test the expected functionality of *guess_number*.

We are going to manually assign the *f_guess param* and set it to *'4'* a *str*. We then manually assign *f_turns_left* to *3* and *int*.

We then create what our expected *return_1* value of *4* and *int* and then *return_2* an int as well.

We then call the actual function and pass in our manual values of *f_guess* and *f_turns_left* and then AFTER the function is complete assign the return values into *integer_1* and integer_2.

Finally we assert that *integer_1* and *return_1* are equal and then we assert that *integer_2* and *return_2* are equal as well.

We then examine this line of code which we will explain below.

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

This means that if you are running this file on its own it will call the **unittest** module. In the web editor do the following.

```
import unittest
unittest.main('test_funcs')
```

You will see that it ran our single test and it passed!

```
Ran 1 tests in 0.001s


OK
```

You will see something along the lines of this which means that we are good to go. We successfully tested the expected behavior of this function and it behaved as we expected.

The next step is to test if our *guess_number* function properly handles an incorrect integer range.

```
    def test_guess_number_incorrect_integer_range(self):
        """
        test guess_number functionality handles incorrect integer range
        """
        # Params
        f_guess = '-5'
        f_turns_left = 3
        # Returns
        return_1 = '\nRULES: Please enter a number between 1 and 9.'
        return_2 = 2
        # Calls
        string_1, integer_1 = guess_number(f_guess, f_turns_left)
        # Asserts
        self.assertEqual(string_1, return_1)
        self.assertEqual(integer_1, return_2)
```

Here we set *f_guess* to *-5* an int and *f_turns_left* to *3* an *int*. We expect that *-5* will cause *return_1* to return the *str* of *'\nRULES: Please enter a number between 1 and 9.'* and *return_2* to return an *int* of *2*.

When we repeat the process of clicking on the green play button in our *if __name__* block we now see 2 tests passing.

```
Ran 2 tests in 0.002s


OK
```

Doing great so far! Let's now test if our function properly handles a non-integer.

```
    def test_guess_number_non_integer(self):
        """
        test guess_number functionality handles non integer
        """
        # Params
        f_guess = 'k'
        f_turns_left = 3
        # Returns
        return_1 = '\nRULES: Please enter a number between 1 and 9.'
        return_2 = 2
        # Calls
        string_1, integer_1 = guess_number(f_guess, f_turns_left)
        # Asserts
        self.assertEqual(string_1, return_1)
        self.assertEqual(integer_1, return_2)
```

Here we pass *f_guess* a *'k'* which is a *str* and we expected to see the same *return_1* value as before. When we run this we see this passes as well.

```
Ran 3 tests in 0.003s


OK
```

Next we test the *did_win* function to check if its expected functionality works.

```
    def test_did_win(self):
        """
        test did_win functionality
        """
        # Params
        f_guess = 5
```

```
        f_correct_answer = 5
        # Returns
        return_1 = 'You won!'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)
```

Here we pass *f_guess* an *int* of *5* manually and *f_correct_answer* an *int* of *5* manually and *return_1* a str of *'You won!'* and then call the *did_win* function and check the actual *string_1* value returned from the *did_win* function with what we expected to be returned with our manual *return_1* variable.

When we run we see success!

```
Ran 4 tests in 0.004s


OK
```

We then test whether *did_win* will handle a proper lower guess value.

```
    def test_did_win_guessed_lower(self):
        """
        test did_win functionality handles guessed lower
        """
        # Params
        f_guess = 4
        f_correct_answer = 5
        # Returns
        return_1 = 'HINT: Higher Than 4'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)
```

Here we have *return_1* return the *str* of *'HINT: Higher Than 4'* and check it against what it actually calls.

```
Ran 5 tests in 0.005s


OK
```

Next we test the opposite.

```
    def test_did_win_guessed_higher(self):
        """
        test did_win functionality handles guessed higher
        """
        # Params
        f_guess = 6
        f_correct_answer = 5
        # Returns
        return_1 = 'HINT: Lower Than 6'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)
```

Success!

```
Ran 6 tests in 0.006s


OK
```

Finally we check for a non-integer.

```
    def test_did_win_non_integer(self):
        """
        test did_win functionality handles non integer
        """
        # Params
        f_guess = 'k'
        f_correct_answer = 5
        # Returns
        return_1 = '\nRULES: Please enter a number between 1 and 9.'
        # Calls
        string_1 = did_win(f_guess, f_correct_answer)
        # Asserts
        self.assertEqual(string_1, return_1)
```

Success!

```
Ran 7 tests in 0.007s


OK
```

Now that you have an idea of how this works please take the time to review the Chocolate Machine in both its functional and OOP version as well as the Escape Room in its OOP version as well at this link.

https://github.com/mytechnotalent/Python-For-Kids

This has been an incredible journey and I am so excited we had the opportunity to share it together!