Anson Learns Python

I'm learnding

- Pybadge Dev Notes
  - Captain's Log: Progress Report
- Learn 2 Python Notes
  - Captain's Log Progress Report
  - Notes - Sunday 12/20/20
  - Notes - Monday 12/21/20
  - Notes - Tuesday 12/22/20
  - Notes - Monday 12/28/20
  - Notes - Wednesday 12/30/20
  - Notes - Tuesday 1/5/21
  - Notes - Wednesday 1/6/21 - Backup of the Temperature Converter Code
  - Notes - Wednesday 1/6/21
  - Notes - Tues 01/12/21
  - Notes - Friday 01/15/21
  - Notes - Sunday 01/17/21

Pybadge Dev Notes

End goal: DIY Tamagotchi

Captain's Log: Progress Report

Learning Python Log

So I can see how I grow because the code doesn’t always show everything I’ve done

Wed 11/11/20

My first day starting to learn Python. Just watched a few intro to Python lectures on YouTube.

Thurs 11/12/20

https://learn.adafruit.com/adafruit-pybadge?view=all#overview

Went through the introductory website for the PyBadge. Got Mu working after I couldn’t figure out Sublime Text Editor. Figured out how to update the firmware and load MakeCode games. Loaded up the intro code to blink the LED. “Hello World”. See 1_One.mp4. Learned what libraries and neopixels are. Found some introductory codes within the libraries: led_animation_rainbow_animations.py was very helpful. See 2_Two.mp4. I managed to get the Neopixels working and I could control the PyBadge through Mu. But I didn’t get any of the buttons working. Created RainbowChase1.py

11/12/20 Notes

- To get into bootloader mode, hit the hard reset button on the back of the PyBadge.
- Neopixels are at D8 on the Pybadge
- Learn more about the NeoPixel coding here = https://learn.adafruit.com/adafruit-neopixel-uberguide/python-circuitpython.

Friday 11/13

Today’s goal was to get the buttons working. I tried using the PyBadge Conference Badge project code: https://learn.adafruit.com/pybadge-conference-badge-multi-language-unicode-fonts/setup. Then I wanted to get and Start and Select working doing something. Needs neopixel.mpy library. Looked over the code a lot to figure out how the Neopixels worked alongside the button pressing. Took the LED parts of the PyBadge Conference Badge code and turned it into Rainbow Chase 2 (RainbowChase2.py). Tried to use this to help: https://learn.adafruit.com/blinka-jump-pybadge-game/coding-the-pybadge. But it was confusing for my current level of knowledge

Figured out

- lines 31-35 are variables
- lines 78-105 was studied a lot to understand how the neopixels turn on and how the buttons work
- Added lines 106-111. Figured out how to alter NEOPIXEL_COUNT and figured out that it’s a variable.

But then was really struggling how to limit the issues. I didn’t know how to set a limit and kept getting a “list index out of range solution” error. Tried a bunch of fancy ways to limit the variable. Figured out that I could just use the limits on lines 102 and 104….only after I realized that “speed” and “brightness” were also variables being controlled. Created RainbowChase2.py. Watch 3_Three.mp4.

Learn 2 Python Notes

Anson learns Python from a Udemy course. Here are some notes.

Captain's Log Progress Report

Anson learns Python

Here's a progress log on what Anson learns on his Python journey. Just to keep track and see progress.

Current Goals

These are taken from notes on 1/5/21

1. Finish the Udemy course on Python
2. Create the game snake
3. Turn my Excel sheet of my video game collection into a SQLite3 database to learn more about Python and databases.
4. Learn about Kivy and Android Studio
5. Port snake to Android
6. Figure out how to turn the SQLite3 video game collection database into an app so I can search through my collection on my phone.
7. Port snake to pybadge and circuitpython

12/20/20 Sun

Starting! My 2 week goal is to create the game Snake in Python. So I can kickstart my new game dev studio.

Learned how Visual Studio Code ("VSCode"), git, and Jupyter notebook worked. Setup note taking log on Phil's Notes Bookstack (Thanks Phil). Created Jupyter notebook for local notetaking for Anson. Created alias to backup files to backup hard drives.

In the Udemy course, I learned:

- Data Types
- Numbers
- Variable Assignments
- Strings
- String Indexing
- String Slicing

12/21/20 Mon

Did some research to see what libraries were available for the PyBadge in CircuitPython. Found code that someone made to create snake in micropython. Wondering if that can translate to my pybadge.

In the course today, I learned about:

- String Properties
- Print formatting with strings
- Lists

12/22/20 Tues

Discovered these Markdown notes from the Jupyter Notebook documentation which were really helpful. And learned that VSCode can do Markdown editing with previews so I'll be using VSCode to edit all my Markdown notes now!

In the course I learned about:

- Dictionaries
- Tuples
- Sets

- Boooleans
- Manipulating files within Jupyter Notebooks and Python 3

Then took the 00-Python Object and Data Structure Basics assessment!

12/28/20 Mon

After a smol holiday break, I'm working on the Udemy course again. Today I learned about:

- Comparison/Boolean Operators in Python
- Chaining Comparison/Boolean Operators
  - And
  - Or
  - Not

12/30/20 Wed

More Udemy coursework. Today I learned about:

- Control Flow - if/else/elif Statements

1/5/21 Tues

New year, same bullshit. Let's learn more Python. Last night I came up with some more goals and ideas:

1. My original goal stands: I want to finish the Udemy course and make the game snake still.
2. I want to learn more about Python and SQLite3 databases. SQLite3 should be a part of Python. So I want to load up my video game collection into a database.
3. I want to turn the database into an Android app using Kivy. So take that database and turn it into a way that I can access the info on my phone.

4. I want to port my snake game into Android as well. Porting the .py file into an .apk.
5. Poems for your sprog message of the day: the idea is that I'd create a python program/script that pulls all of /u/poem_for_your_sprog's poems and comments on reddit and displays them as a message of the day when you open up your terminal. Would also be really fun as a bot on discord to get a random sprog.
    - Reddit idea
    - GitHub

Ubuntu Machine - Installing Jupyter Notebooks and Python

I switched to my Ubuntu machine today. It didn't have pip, anaconda, jupyter, or anything that allowed the Jupyter notebooks to run Python. I kept getting "Data Science libraries notebook and jupyter are not installed in interpreter Python 3.8.5 64-bit." as an error. To fix this, I had to install pip, anaconda, ipykernel, and jupyter. Anaconda came as a script from the anaconda website. To install scripts, I just dragged the script file in the GUI into the terminal. Then I ran the these terminal commands. Remember updateme is my alias that does `sudo apt update && sudo apt upgrade -y && sudo apt autoremove -y && sudo apt autoclean -y && sudo snap refresh`.

```
updateme
python3 -m pip install --upgrade pip
python -m pip install ipykernel --force
python -m pip install jupyter --force
```

In terms of the Udemy course, today I learned about:

- For Loops

1/6/21 Wed

GDQ and trying to learn Python is hard but let's make a little progress today!

In terms of the Udemy course, today I learned about:

- While Loops

- Keywords for Loops - Break, Continue, and Pass
- Range
- Enumerate
- Zip
- In
- Min/Max
- Random Library
- Input
- List Comprehension

Temperature Converter Program

See this link for more information

01/12/21 Tues

More Udemy coursework after a break due to the holidays and AGDQ 2021. Today I learned about:

- Part 2 Testing - checked my understanding for list comprehension and for/if/elif statements
- Methods and Python Documentation aka Python help
- Introduction to Functions
- def keyword
- Python Functions - Basics, logic, tuple unpacking, interactions

01/17/21 Sun

New twist to the Udemy course: Let's work on function practice problems.

- The practice problems I'll work on are found here: ~/Nextcloud/L2Python/'Udemy Python Class'/'03-Function Practice Exercises-AW-011721.ipynb'.
- They're also in the notebooks from the course: ~/Nextcloud/L2Python/Udemy Python Class/Complete-Python-3-Bootcamp Notebooks/03-Methods and Functions.

Notes - Sunday 12/20/20

Starting! My 2 week goal is to create the game Snake in Python. So I can kickstart my new game dev studio.

I got overwhelmed with options for editors and went with Visual Studio Code ("VS Code") which was recommended by the jwaz nerd chat. Mu will be my backup as I'm currently not sure how VSCode will work with the Pybadge or how to get a REPL working.

Below is an example code we were to use to teach us how to save a .py file in Sublime or another text editor.

print('hello world')

Jupyter Notebooks Notes

How to open files

I can open Jupyter notebooks through the actual Jupyter Lab program by running `jupyter-lab` in the CLI. That will open something up in the browser which I can navigate around in. Or I can open the notebooks up in VSCode which I like since the one app can open up my code, my own notebooks, and the notebooks from the class.

How to create new notebooks in VSCode

`Ctrl+Shift+P` then type in "Create new blank Jupyter notebook

Git Notes

Once I start getting more complicated projects, I'll want to keep backups or setup "source code management" (SCM) to track changes to my code. Git helps to do this and is seperate from Github. Git can be a local tool and I can back up my projects and code to my Nextcloud server if I want. Github is a tool that uses git but is onilne and I could share projects with others among other features.

As of 12/20/20. I haven't setup git yet. I should probably do so in the future.

Some helpful links:

- VSCode documentation
- Colt Steele's Learning about Git in 15 mins
- Colt Steele notes to go with Youtube video above
- Digital Ocean CheatSheet

Types of Data

- Integers (int) = Whole numbers - 3, 300, 200
- Floating Point (float) = Numbers with decimal places - 2.3, 4.6, 100.0
- Strings (str) = Ordered sequence of characters so just text - "hello", "Sammy", "2000"
- Lists (list) = Ordered sequence of objects so text, Integers, or Floating Points could go in here - [10,"hello",200.3]
- Dictionaries (dict) = Unordered Key:value pairs (more info coming later; currently doesn't make sense to me) - {"mykey":"value", "name":"Anson"}
- Tuples (tup) = It's said like "too-pul-es". It's a sequence of objects that and the order can't be changed aka immutable (more info coming). - (10,"hello",200.3)
- Sets (set) = Unordered collection of unique objects {"a","b"}
- Booleans (bool) = Logical value indicating True or False

Note that:

Integers do not have decimals.

A Floating Point number can display digits past the decimal place. A floating point number does not represent EXACT values, just a close/increasingly better approximation of a value. For example 1/3 doesn't mean the value is 0.3, 0.33, or 0.333. It's how the computer holds the approximate value.

So 0.1+0.2-0.3 doesn't equate to 0.0. Just a value really close to 0. Click this link for more info.

Numbers in Python

There's 2 main types of numbers as listed above: integers and floating point.

Python can be used as a straight up calculator. I can literally type in 2+2 and run the code to get a value.

Mod operator %

You could also have python output a "Mod operator" or a "Modulo".

Let's say you wanted to do 7/4. You would say that 4 goes into 7 one time with a remainder of 3. So the mod operator will output the remainder. In this way, you can quickly see if a number is easily divisible by another number.

In the previous example, we had a remainder of 3. So 4 does not go into 7 easily. But 5 goes into 50 cleanly. We would expect the remainder to be 0.

To use the mod operator, you'd use the % sign. `X % Y`

You can also use the mod operator to check if a number is even or odd. For example if you have a variable that is a number and you're not sure if it's even or odd. If I get a remainder from the mod, then it's odd. If I get 0, then it's even.

23 is odd. 20 is even.

Exponents **

I can do Exponents by doing `x ** y`.

Order of Operations

Remember PEMDAS for order of operations - parenthesis, exponents, multiplication/division, addition/subtraction

If I just type in normal formula. Python will follow the order of operations. If I want to control the order Python does the math in, I will have to use parenthesis.

Variable Assignments

We could assign the above data types a variable name.

Names can't:

- Start with a number
- No spaces - need to use underscore _ instead
- No special symbols besides _ (Python will complain if the special symbols are used)

Names should:

- Be in lowercase as a best practice
- Avoid keywords as variable names such as list or str (the editor should highlight those keywords a different color)

Python uses dynamic typing. So variable names can be reassigned. The below can happen. In other langauges such as C++, this can't be done. Once the integer of "my_dogs" is assigned, it stays that way.

- `my_dogs = 1`
- ``my_dogs = ["Armani"]

Dynamic Typing Pros:

- easy to work with
- saves time since you don't have to assign a data type. So for example, in the above example, C++ wants to see an integer data type assignment. `int my_dogs=1`

Dynamic Typing Cons:

- Easy to have unexpected data types could cause bugs

How to determine data type using Type()

Let's say we don't know what kind of data "a" is and we want to find out. We would use `type(a)`

- type() could help find those errors (discussed below)

I ran the code below in a Jupyter notebook.

```
a = 5

type(a)

a = 30.1

type(a)
```

Syntax Highlighting and variables

Note that above we can see that type() shows that my data was an integer then a floating point number. We can also see that "type" is highlighted in an aqua green color. That shows that its a

python keyword. So we shouldn't name a variable "type". If I accidently make a variable named "type", I can undo this by restarting the variable. In VSCode, the symbol is the little green circular arrow.

Variable name example

Let's say I want to calculate my taxes.

- Income = 100
- Tax Rate = 0.1

```
my_income = 100


tax_rate = 0.1


my_taxes = my_income * tax_rate
```

Then run `my_taxes` and the answer should be 10.0

Strings

Strings are sequences of characters. The syntax used is single or double quotes.

- 'hello'
- "hello"
- " I like pie "

Strings are ordered sequences. So we can index and slice to grab parts of the string. The spaces/whitespace also count as characters. The syntax is ' or " so that if there's an abbreviation for example, you can include the abbreviation without confusing python. So for example the string would be:

- "I'm a lover of pie"

Instead of:

- 'I'm a lover of pie' Which would generate an "invalid syntax" error if the code was run.

Indexing

Indexing allows you to grab a single character from the string. It uses [] notation after the string or string variable. If my string is hello:

- 'hello'
- Characters = h e l l o
- Index show = 1 2 3 4 5

Reverse index allows me to grab the last letter of hello even if I don't know how long the string is.

- Reverse Index = 0 -4 -3 -2 -1

Slicing

Slicing allows me to slice a part of the string to grab a subsection of multiple characters. The syntax would be:

- [start:stop:step]

Where:

- start = numerical index for the start of the slice
- stop = index I'll go up to but not include
- step = size of the slice we'll take (size of the jump)

```
'hello world'
```

So in the above examples, we're asking Python to return the string we type in. That's why we see the little quotes. But we could also print the strings out. This removes the quotes and actually prints out the string. If we wanted to print multiple strings for example, we would need to use the print command.

In the print command, we could also use escapes `\n` or tabs `\t` to change the formatting of the thing we print out. Syntax would be `print("hello \n world"` and `print(hello \t world)` respectively. More on this coming up.

We can also check the length of the string using the `len` command. Syntax is `len('x')`

String Indexing and Slicing - Let's look at it more in depth Indexing in more depth Let's set a variable called mystring to be a sting that says "Hello World". The syntax for using a string would be mystring[position]. The positions start at 0, NOT 1. When we type in the variable, this is referred to as "calling". So below I'll set the mystring variable then call it and run an index to get the first letter H which is in position 0. Typing in the 0 would be called "passing".

If the position 0 gives us H. The index would count left to right in positive integers. So let's say I want the capital W. I would pass 7 in the index to get it.

Let's say I want to have "d" be the result of calling the index. I could use the normal index and pass 10. Or I could use the reverse index. If I want to use a reverse index, I'd count from right to left from the end of the string. So I would pass -1 to get the "d" or -5 for the "W".

Slicing in more depth

Let's say now `mystring = 'abcdefghijk"`. I need to redefine mystring. Then use syntax `mystring[position:]` . The : shows where the silce should start or end. I want to grab a slice or chunk of the string that's C through to the end.

So to grab C through to the end I'd type `mystring[2:]`.

If I want to grab "abc", I'd write `mystring[:3]`.

I can also grab a slice in the middle so if I want "def" I'd type `mystring[3:6]`

Remember that you're going UP to the position but NOT including that position. So that's why I'm including position 3 which id "d" in my string but it won't print out "abcd" but instead prints "abc".

Notes - Monday 12/21/20

Day 2 of working on the Udemy Python Bootcamp course! We're going to pickup with more info about strings.

String Properties and Methods

Strings are "immutable" or cannot be changed. So for example:

- If I set `name = "Bob"`
- And then I try to set 'B' to 'R', I'm not allowed to. I'll get an error.

Concatenate

If I do want to change that 0 position, I will have to use concatenation.

1. Slice out the parts of the string I want to keep by creating a new string based on the first string. So in this case, I wan't to create the string "last_letters" based on "name".
2. Concatenate using + so I would take the "last_letters" and then Concatenate "R" to make Rob.

- `last_letters = name[1:]`
- `'R' + last_letters`

Using Concatenate is a way to put strings together. I can also add a variable to a string and run the cell to generate a longer statement.

```
x = 'Hello World'
x = x + ' it is nice out`
x
```

So the first run would say "Hello World it is nice out". And running the cell again would add on more "it is nice out". Concatenate could be added or multiplied. Remember that it's easy to confuse integers and strings. This is a con of the flexibility of the dynamic typing in Python. For example:

- `2+3` will yield 5
- `"2"+"3"` will Concatenate the strings "2" and "3" and yield 23

Functions

Variables can be modified with Python built in functions. First, define the variable as a string. In my example, I want `x = 'Hello World'`.

Now if I type x. in VSCode, a list of functions will appear. Syntax for these functions are `variable.function()`. If you don't include (), then Python will think you're just asking what that function will do.

Example of functions include:

- upper for all upper case letters in the string
- lower for all lower case letters in the string
- split will take your string and turn it into a list. The default split will divide the string at each whitespace. I can tell split where to split the list up.

Important Note: Comments

To add comments to my code, create a line that begins with # then write the comment out

String Formatting for Printing - String Interpolation

There are situations when you want to put a variable or objects ("inject") into a string so you can print it. For example:

```
my_name = "Jose"
print("Hello" + my_name)
```

There's 2 methods to do this:

1. .format()
2. f-strings (formatted string literals)

.format()

The .format() syntax is: `print('urnormalstringhere {}'.format('urobject'))`

You can add in multiple objects and multiple {}. The syntax would be: `print('urstringhere {} {} {}'.format('object1','object2','object3'))`

And within the {} you can assign a position number and then set an order based on the positions in the format string. The syntax would be: `print('urstringhere {0} {1} {2}'.format('object1','object2','object3'))`

You can also assign the object keywords and then set an order based on the keywords. The syntax would be: `print('urstringhere {x} {y} {z}'.format(x='object1',y='object2',z='object3'))`

Floating point number formatting

Let's say we have a variable that's a division problem. And then we want to print the answer from the division problem in a cleaned up format.

So the example code would be:

```
result = 100/77
result
```

Then we want to print the result but just with 3 decimal places. The syntax would be: `{value:width.precisionf}`

So the print code would look like:

```
print("The result was {r:1.3f}".format(r=result))
```

Where value r is the variable result. And then it's passed inside the {} to adjust the float formatting.

- Value is the variable we set in the .format()
- Width is how many white spaces you want to show.
- Precision (f) is how many decimal places do you want to show
- If you don't want the precision decimal places and just want to set a number of positions, leave the f out.

f-string String Interpolation

f-string is a simplified version of the .format(). So Let's say my variable is `name = 'Bob'` and I want to print a statement that says "Hello his name is Bob".

The example code would be:

```
name = "Bob"
print(f'Hello, his name is {name}')
```

So the syntax is `print(f'urstringhere {urobjecthere}')`.

Multiple objects can be used in an f-string. And .format() and f-string can be combined.

More reading can be found here:

- pyformat
- f-strings

Lists

`list = [1,2,3,4,5]`

Lists are ordered sequences that can hold objects. Could be a list of integers or a mixed list of data types. They use [] brackets and commas to seperate objects in the list. Syntax is `[x,y,z]` but we probably want to name the list or assign it a variable. So the syntax would be `variablename = [x,y,z]`.

Various methods can be called off them. Some things you can do with lists:

- indexing (this should feel like manipulating a string)
- slicing (this should feel like manipulating a string)
- nesting lists
- check the length using `len()`
- concatenate lists together
- add functions (remember to include () to have the functions work and not have Python tell you what things are)
  - Append adds a new object to the end of the list which is called affecting the list in place - syntax would be `list.append()`
  - Pop removes objects off the list or by index location. Index location can be in normal index order or reverse index. - syntax would be `list.pop(x)` (leave x out if you want the last object taken away)
  - Sort will put your list in order. Note that this is done in place. The sorted output needs to be assigned a variable in the next line after the sorted output happens in order to save the sorting. - syntax is `list.sort()`
    - To save the sorted output do the following:

      ```
      list.sort()
      sorted_list = list
      ```

    - Don't do:
      `sorted_list = list.sort()`

Unlike a string, lists can be changed or are mutable. So If I have a list showing 1-5, I could change 1 to ONE by using:

```
list[0] = "ONE"
list
```

Notes - Tuesday 12/22/20

Day 3 of Python Bootcamp LFG.

Dictionaries

Dictionaries are unordered mappings for storing objects. Mappings are collections of objects that are stored using a key. A sequence stores objects using the position. Mappings won't have an order and can't be sorted. They can't be indexed or sliced.

There are dictionaries that can be ordered called `ordereddict` which will be covered later.

Lists store objects in an ordered sequence. Dictionaries store objects using a key value pairing. The value is a Python object. This value pairing allows the user to grab the object without knowing the index location. Retrieving something from a dictionary can be faster than getting it from a list.

The syntax is:

```{'key1':'value1','key2':'value2'}```

Note the single quote ' is used.

To retreive something from the dictionary, you'd type in the dictionary name then pass the value you want. The syntax is:

`urdictionary['urkeyhere']`

Dictionaries can hold (among other things):

- Lists
- Floating point numbers
- Strings
- Integers
- Other dictionaries

I can call dictionaries within dictionaries or list index positions within dictionaries. I would first retreive the object as seen above with the basic command `urdictionary['urkeyhere']` but then I'd addon the second object or command I'm trying to pass. For example, if I wanted to retrive an index from the dictionary then call out index position 2, my command would look like:

`urdictionary['urkeyhere'][2]`
```

If I want to add to an existing dictionary, I would simply call the dictionary I previously defined, then add a new key. The syntax would be:

`urdictionary['urnewkeyhere'] = urnewobjecthere`

This command could be used to create a new object and key in the dictionary or modify an existing key with a new object.

Dictionaries have functions. Some examples are:

- Keys - Shows all the keys in the dictionary
- Values - Shows all the values aka objects in the dictionary
- Items - Shows everything in the dictionary but as a tuple ("Too-pul") which will be discussed in a later topic.

The syntax would be: `dictionary.function()`

Remember like before, remember to include () or else you're asking Python to tell you what the function could do. Instead of actually executing the function.

Dictionary Manipulation Exercise

Let's do a basic example of manipulating a dictionary that has a list of letters

`list = [a,b,c,d]`

Let's say I want to retreive "c" then save it as a variable and capitalize it.

The long way looks like:

```
1 alphabet_dict = {"k1":['a','b','c','d']}
2 alphabet_dict
3 # Set up the dictionary and called it to confirm it's good.
4 alphabet_list = alphabet_dict['k1']
5 alphabet_list
6 # Set the variable alphabet_list to be the k1 key of the dictionary then called it to confirm it's good.
7 letter_c = alphabet_list[2]
8 letter_c
9 # Set the variable to pull "c" from the variable alphabet_list
10 letter_c.upper()
```

```
11 # Used function upper to capitalize "C"
```

Or you could stack calls on top of each other to make it easier. Python allows this flexibility. So easier:

```
1 alphabet_dict['k1'][2].upper()
2 # Did work from above lines in one step:
3 # First we re-use the alphabet_dict dictionary that was defined
4 # Second we call the key 'k1'
5 # Third we call index position 2
6 # Last we use function upper to capitalize that value
```

Tuples

Say "too-puls"

`tuple = (1,2,3,4,5)`

Tuples are like lists except they can't be changed ("immutable"). Once an object is inside a tuple, it can't be changed. The features and manipulation of a tuple is like a list but not as flexible. But tuples have count and index functions.

Tuples are often used for objects that are passed between things and it keeps your data safe and not over-written.

Sets

Sets are unordered collections of unique elements. There's only one representative of that object. So "a" or 2 can only be in the set once.

Syntax is:

```
1 ursethere = set()
2 ursethere.add(urobjecthere)
```

We'll learn more about sets later in the course. But a use for sets right now is you could take a list of numbers and get a unique list of the values.

So if I have a list of random numbers, casting the list into a set will show me the unique numbers in that set.

```
1 list_num = [4,4,4,4,4,3,3,3,3,6,7,7,7,8,89,9]
2 set(list_num)
3 {3, 4, 6, 7, 8, 9, 89}
4 # Line 3 is the output of Line 2
```

Booleans

Booleans are operators that allow you to convey True (aka integer 1), False (aka integer 0), or None. statements. They're really helpful when dealing with control flow and logic. Make sure that your boolean is Capitalized. Otherwise Python thinks you're creating a variable.

For example it plays a part in comparison operators like > and ==. If I ask Python if `1 > 2` (Is one greater than 2?) or `1 == 1` (Is one equal to 1?), it will output a boolean answer.

Files - Simple Input/Output with Files in Jupyter Notebooks

The following notes all only work in Jupyter Notebooks which came from iPython notebooks.

In Jupyter Notebooks, in a cell I can directly create a text file. The command is `%%writefile`. The syntax is `%%writefile urfilename.txt```. Then on the second line after `%%writefile``` I can start writing the actual text I want to add to the file.

I could then create a variable to open the file. If I get a `Errno 2` when I try to open the file, I spelled the file name wrong or I'm pulling from the wrong directory. The syntax would be `urvariable4filename = open('urfilename')`.

To check your current directory, type `pwd` into a Jupyter notebook cell. Just like how you would check the current directory in the terminal.

If I opened the file with a variable, there's functions I can do. It's similar to the functions we previously covered before with the `urvariable4filename.function()`. And of course, remember the (). These functions include:

- Read - it will list out the entire contents of the file
  - \n means new line which is an escape sequence which was previously covered
  - Running read will move the cursor to the end. To reset the cursor, you need to use Seek.
- Seek - this is how you reset the cursor in read. Syntax is `urvariable4filename.seek(0)`

- Read Lines - it will read through the lines in a more legible manner and you can perform other things to this readlines function. Syntax is `urvariable4filename.readlines()`
- Open - This just opens up the file that you want to work on. Simply using this open method will require that you close the file.
- Close - Remember to close out your files so that the computer doesn't think Python is still using the file.

To open files in another location, you have to change the file path when running open.

- For Linux the syntax is: `myfile = open("/Users/YouUserName/Folder/myfile.txt")`
- For Windows the syntax is: `myfile = open("C:\\Users\\YourUserName\\Home\\Folder\\myfile.txt")`

To open files AND then set modes and do something with the files, I'll want to use `with open` so the syntax would be:

```
with open ('urvariable4filename',mode='urmodehere') as urvariablehere:
    whatyouwannadohere(urvariablenamehere.function())
    # note the indentation
```

You could use this method to:

- Read what's in the file using in the indented line `print(urvariablenamehere.read())`
- Append the file or add on to it using in the indented line `urvariablenamehere.write('urtexthere')`
- Write the file using

%%writefile Modes

- Read only = mode='r'
- Write only - will overwrite or create new files = mode='w'
- Append only - add onto existing files = mode='a'
- Reading and Writing = mode='r+'
- Wrtiring and Reading - will overwrite or create new files = mode'w+'

Notes - Monday 12/28/20

12/28/20 Monday

Let's keep going with the Udemy course! We took a smol holiday break. Starting with boolean operators.

Boolean Comparison operators

== Equality: Checks to see if objects are equal to each other. Type of data matters so if you compare `"2" = 2` it won't yield True since you're comparing a string to an integer. If you compare `'Bye' = 'bye'` it won't yield true since case matters.

!= Inequality: Checks to see if objects are UNequal to each other. So let's say you want to see if 4 is not equal to 5. Your syntax would be `4 != 5` and Python would yield True.

`>` Greater Than: Checks to see if the objects are greater than each other.

< Less Than: Checks to see if the objects are less than each other.

`>=` Greater Than or Equal to: as described. Like the greater than and equality check together.

<= Less Than or Equal to: as described like the less than and equality check together.

Chaining Boolean/Comparison Operators

So we discussed comparison operators above:

- Equality ==
- Inequality !=
- Greater Than >
- Less Than <
- Greater Than or Equal to >=
- Less Than or Equal to <=

Now we're going to add in:

- AND
- OR
- NOT

Let's say that we want to compare 1, 2, and 3.

1. `1 < 2 < 3` will work and output True.
2. `1 < 2 > 3` will output False even though `1 < 2` is true.

AND Operand

In example #2, we could use the `AND` operator which would link the two comparisons together. So in other words, `AND` wants both the statements on either side to be True. The syntax would be:

```
1 < 2 AND 2 > 3
```

So this is checking if `1 < 2` is true AND THEN if `2 > 3```` is true. If both are true then we'll get an output of true. But since `2 > 3``` is false, the result will output as False.

OR Operand

This logic links 2 statements together and checks to see if either pass the logic presented. In other words, `OR` wants one of the statements on either side to be True. So if we use example #2 again, the `OR` operator can be used. And the syntax would look like:

```
1 < 2 OR 2 > 3
```

So this is checking if either of the comparison statements are true. Either statement on the left or right can yield a true output and the `OR` operator will output as True.

NOT Operand

This logic looks for the reverse of what's inside the statement. Another way to say this is that the `NOT` operand is looking for the opposite of the boolean or syntax or statement in the line of code. So let's say our example is:

```
100 == 1
```

So if we ran this as is, we would expect a False output as 100 does not equal 1. But if we used the `NOT` operand then we would expect a True output. This is because we're asking Python if the example statement is NOT equal. So the syntax would look like:

```
not 100 == 1
```

Future Usage

These `AND`, `OR`, and `NOT` operands can be used in these simpler boolean/comparison operators. It could also be used in If statements.

The syntaxes might change depending on the library used but these operands could be wrapped in a parantheses or not. For example the above `NOT` operand example could look like either one of these:

- `not (100 == 1)`
- `not 100 == 1`

For now either of these are acceptable but as stated, the syntax may change depending on the libraries used which will be covered later.

Notes - Wednesday 12/30/20

12/30/20 Wednesday

I started Python statements on 12/28/20 Monday but didn't get very far. So I'm including it here.

Python Statements

Let's learn about stuff like If then statements. I don't know a lot of statement examples lol...

if, elif, and else statements

These statements will introduce the idea of control flow. This is where we build a lot of code but only want certain parts of the code to execute depending on certain conditions. For example, `if` my dog is hungry `then` I will feed the dog. This code will be executed in this condition but otherwise the code will move on to the next part if the condition isn't met.

Control Flow Statements

Some keywords used in control flow are:

- if
- elif
- else

These statements use syntax that takes advantage of whitespaces/indexes and colons `:`. The whitespace/indexes are unique to Python and how Python

if statements

if Statements are self explanatory. They begin the block of code that you want to be executed in a certain conditions or situation. Below is the basic syntax. Note that line 2 is indented. Line 2 will be executed when the if statement in line 1 is True and met.

```
1 if some_condition:
2   execute_some_code_here
```

if/else statement

It's like a juiced up if statement. The if statement only executes when the condition is met. And then that's it. if/else will execute when the condition is met otherwise it will do something else. That's why it's called if/else. Below is the basic syntax. Note the lines are indented.

So if the some_condition is met in line 1, then the line 2 code will be executed. Otherwise line 3 will trigger and then the line 4 code will be executed and done. The line 4 else line doesn't have a condition associated with it. This is because it relies on the line 1 condition being True.

Syntax-wise, note that the if and else in lines 1 and 3 are lined up.

```
1 if some_condition:
2   execute_some_code_here
3 else:
4   execute_some_more_code_here
```

if/else statement with elif statement

In the if/else statement above, if you want to check for other conditions before the else statement kicks in, you can add in some elif statements. elif is Else If so you can stack if statements.

```
1 if some_condition:
2   execute_some_code_here
3 elif some_other_condition:
4   execute_some_other_code_here
5 else:
6   execute_some_otherother_code_here
```

Examples on how the above statements would work

if statement

Let's say that we want our code to have a variable called 'hungry'. If it's true, then I want to print out 'Gimme food!'.

So our example would look like the below. And when the code is run, we would get an output of `Gimme food!`. If we set 'hungry' to false, there's no output. This is because the if statement isn't activated so there's no expectation to see anything.

```
1 hungry = True
2
3 # hungry set to True would output "Gimme food!"
4 # hungry set to False wouldn't output anything
5
6 if hungry:
7    print("Gimme food!")
```

if/else statement

Now with the above statement, let's say that that when hungry = False, we want to see an output of `I'm not hungry`. This is so that when the False condition is met, we get some kind of output instead of nothing like above.

```
1 hungry = True
2
3 # hungry set to True would output "Gimme food!"
4 # hungry set to False wouldn't output anything
5
6 if hungry:
7   print("Gimme food!")
8 else:
9   print("I'm not hungry")
10
11 # see how the else statement is aligned with if and print is indented
```

if/elif/else Statements

Let's say I want to make a shitty guide around town about places. I want to ask the bot about locations and have them described to me.

Locations:

- Auto shop - "Car library"
- Bank - "Money storage"

- Other not listed locations - "That option isn't available"

I can setup an equality statement to check and see if my variable is 'bank' or "auto shop'.

Example if/elif/else commented code:

```
1 location = 'Bank'
2 # location is set to bank as a variable
3
4 if location == 'Auto Shop':
5   print('Car library')
6 # I'm checking with an equality statement to see if the location is Auto Shop here. If the location
7 is Auto Shop, then the description will be "Car library".
8
9 elif location == 'Bank':
10  print("Money storage")
11 # I'm checking with an equality statement to see if the location is Bank here. If the location is
12 Bank, then the description will be "Money storage".
13
14 else:
15  print("That option isn't available")
16 # I'm checking with an equality statement to see if none of the defined locations are listed, the
17 code will output a generic "That option isn't available.
```

Example if/elif/else code w/ no comment:

```
location = 'Bank'


1 if location == 'Auto Shop':
2   print('Car library')
3 elif location == 'Bank':
4   print("Money storage")
5 else:
6   print("That option isn't available")
```

Remember that we can stack elif statements so for example, let's say we want Library to output "book storage"

```
location = 'Bank'


1 if location == 'Auto Shop':
```

```
2   print('Car library')
3 elif location == 'Bank':
4   print("Money storage")
5 elif location == 'Library':
6   print("Book storage")
7 else:
8   print("That option isn't available")
```

Notes - Tuesday 1/5/21

1/5/21 Tuesday

New year, same course. Let's learn more about Python.

For Loops

Many objects in Python are iterable or are capable of being repeated. This means we can iterate over every element in the object. For example, we can:

- Iterate or repeat every element in a list
- Each character in a string
- Every key in a dictionary

An associated term is ***iterable*** which means the object can be iterated.

We can use For Loops to execute a block of code for each iteration.

Basic Syntax looks like:

```
1 variable_2_iterate = [1,2,3]
2 for my_variable_name_4_item in variable_2_iterate
3    print(my_variable_name_4_item)
```

So in this for loop, I'm saying that if I see this variable, it's a stand-in for the list of numbers. When I see the list of numbers, I want it to print out the list of numbers. But I could also just say that when I see that list of numbers, I want it to print out "Hello World". So let's change my example:

```
1 for_loop_list1 = [1,2,3,4,5,6,7,8,9,10]
2 for bananas in for_loop_list1:
3    print(bananas)
```

If line 3 says to print `bananas`, it will just output the list of numbers I had in for_loop_list1. If I said for it to print hello world, I'd get Hello World back 10 times since I have 10 items in the list.

```
1 for_loop_list1 = [1,2,3,4,5,6,7,8,9,10]
2 for bananas in for_loop_list1:
3    print('Hello World')
```

Advanced Example 1

Take note of the indentations.

```
1 for num in for_loop_list1:
2     # Check for even numbers
3     if num % 2 == 0:
4     # This says I'm setting variable to num then
5     # checking that when I divide by 2, the
6     # remainder is 0
7         print(f'Even Number: {num}')
8     else:
9         print(f'Odd Number: {num}')
10     # f-string literal to make the printing look nicer
11     # so I'm setting up what I want it to say then putting
12     # variable 'num' in the {}
```

Advance Example 2

Let's look at iterations in strings now. The below example will print out each character in `'Hello World'`.

```
1 mystring = 'Hello World'
2 for letter in mystring:
3    print(letter)
```

Note that you don't have to set any variables if you don't want to. Using `_` is useful for when you want the code to do something but you don't intend on using the block of code again or don't need the variable in the future.

No variable for the string:

```
1 for banana in 'Hello World':
2    print(banana)
```

No variables at all:

```
1 for _ in 'Hello World':
2   print('Fuck you')
```

Tuples in a For Loop

Tuples have a special feature called Tuple Unpacking. This is commonly used in Python libraries. There will be tuples inside a list. If you call them in a for loop, the tuples in the list will be repeated as the objects being iterated on. But you could also further unpack them into their individual objects. So the For Loop could also pull out one object in the tuple. It does this by repeating the tuple as a packed object and then you can unpack it to a more granular level. So the syntax would be:

```
1 # your_data_here - could be a list or a tuple or a string
2 for ur_variable_name in your_data_here:
3   print(variable_u_want_2_print)
```

So an example might be:

```
1 tuple_list = [(1,2,3),(4,5,6),(7,8,9)]
2 for x,y,z in tuple_list:
3   print(y)
```

With your output being: 2,5,8.

Dictionaries in For Loops

You can setup dictionaries in For Loops and they have the same tuple unpacking features. Remember that dictionaries are NOT sorted. The default output in dictionaries is the key. The syntax would be as seen below. And you'd get the keys as the outputs.

```
1 ur_dictionary = {'k1':1,'k2':2,'k3':3}
2 for item in ur_dictionary:
3   print(item)
```

Let's say I want to see both the keys and the values. I have to modify the code in line 2 by adding `.items()` to the end of object.

```
1 for_dict = {'k1':1,'k2':2,'k3':3}
2 for item in for_dict.items():
3   print(item)
```

Let's say I want to see the tuple unpacking in action. I have to modify the code in line 2 by adding the variable setup so `key` would be the left value and `value` would be the right value. Then I'd set `.key()` to the end of object. This would output the key. And I can run it for value by swapping in `.value()` in line 2 and `value` in line 3.

```
1 for_dict = {'k1':1,'k2':2,'k3':3}
2 for key,value in for_dict.key():
3   print(key)
```

Notes - Wednesday 1/6/21 - Backup of the Temperature Converter Code

1/6/21: My first Python Code. It converts temperatures from F to C and C to F.

Huge thanks for @Tokugero and @Leeoku for helping me. And Girlshark for the inspiration

Code with comments

```
# Instantiate initial true flag to enter loop
run_loop = True
# Set global variable to count
retry = 0
# Runs code while retry is under 3
while retry < 3:

    # While set to go
    while run_loop:
        # Try to capture a float at input time so we don't have to parse it later
        try:
            temp = (float(input("Enter temperature = ")))
        except Exception:
            # Exception catches all errors
            # more info here: https://docs.python.org/3/library/exceptions.html
            print("Input isn't a temperature; try again. Max 3 attempts. Attempts:",retry+1)
            retry += 1
            # If we can't establish a float for the first input, we'll simply skip the rest of this iteration and never set
            # the run_loop flag to false, allowing loop to continue
            # Print is setup so it tells me how many attempts I'm at and shows the count
            break
            # Ends the program if I guess too much
```

```
        # Instantiate a sub loop
        run_loop_sub = True

        while run_loop_sub:
            try:
                unit = str(input("Enter C or F (for Celsius or Farhenheit) = "))
            except Exception:
                print("Input needs to be c/C or f/F")
                # Harder to hit this since "" is a string in input, but if it fails for whatever reason
                # just try again
                continue

            if unit.lower() == "c":
                fahrenheit = (temp * 9/5) + 32
                print(f'{round(fahrenheit,2)} F')
                print('You know the temp now!')
                # Completion condition met, set loop flag to false to exit loop after this iteration
                run_loop_sub = False

            elif unit.lower() == "f":
                celsius = (temp - 32) * 5/9
                print(f'{round(celsius,2)} C')
                print('You know the temp now!')
                # Completion condition met, set loop flag to false to exit loop after this iteration
                run_loop_sub= False

            else:
                print("You need to enter either c/C or f/F")
                # There is no satisfactory completion here, so don't set the close flag

            # If we make it here, that means that the sub while loop was satisfied, and there is no further exceptions
to
            # skip this flag; we can probably end the loop
            run_loop = False

    #if retry == 3:
    #    run_loop = False
        # Not sure if this helps or hurts
        # After experimenting, it doesn't seem to matter if it's here
```

Code without comments

```
run_loop = True
retry = 0
while retry < 3:

    while run_loop:
        try:
            temp = (float(input("Enter temperature = ")))
        except Exception:
            print("Input isn't a temperature; try again. Max 3 attempts. Attempts:",retry+1)
            retry += 1
            break

        run_loop_sub = True

        while run_loop_sub:
            try:
                unit = str(input("Enter C or F (for Celsius or Farhenheit) = "))
            except Exception:
                print("Input needs to be c/C or f/F")
                continue

            if unit.lower() == "c":
                fahrenheit = (temp * 9/5) + 32
                print(f'{round(fahrenheit,2)} F')
                print('You know the temp now!')
                run_loop_sub = False



            elif unit.lower() == "f":
                celsius = (temp - 32) * 5/9
                print(f'{round(celsius,2)} C')
                print('You know the temp now!')
                run_loop_sub= False

            else:
                print("You need to enter either c/C or f/F")
            run_loop = False
```

Notes - Wednesday 1/6/21

1/6/21 Wed

Other Useful Operators and Functions

Range Operator

Often, we'll have lists of numbers and instead of typing out all the numbers in the list, we can use the Range function. This works similar to the slicing function mentioned above. So let's say I want to print out all the numbers in a range. I could setup a list:

```
num_list = [0,1,2,3,4,5,6,7,8,9]
```

Or I could use range. Remember that like in a slice, the range ends one value before the one you define. So if I put 10 into the range, Python counts up to 9. The basic syntax is:

```
range (number-1)
```

You can also specify where you're starting the range. Syntax is:

```
range (start, end-1)
```

You can also add in a step. Syntax is:

```
range (start,end-1,step)
```

Or you can get a list of numbers with a range using:

```
list(range(start,end-1,step))
```

Enumerate Function

Takes any iterable object and returns an index counter and the object or elements. So let's say you want to keep track of how many times a Loop runs. This is better explained with examples:

This is the example before I learned `Enumerate`:

```
# enumerate function
# I want to count the loops happening
# I could setup this counting thing
# but we do this enough in Python that we have enumerate

index_count = 0

for letter in 'abcde':
    print('At index {} the letter is {}'.format(index_count,letter))
    index_count += 1
```

Now let's simplify the example with `Enumerate`

```
# This counts the loops using enumerate and outputs tuples

index_count = 0
word = 'abcde'

for letter in enumerate(word):
  print(letter)
```

Zip Function

This puts lists together. The basic syntax is `zip`. So the objects in the list will be determined by the list with the least amount of functions. The output is a tuple but the tuple only shows if you run something. Otherwise it acts like a normal list. You can also do tuple unpacking.

Again this makes more sense looking at examples:

```
zip_list1 = [1,2,3,4,5,6]
zip_list2 = ['a','b','c']
zip_list3 = [100,200,300]

for item in zip(zip_list1,zip_list2,zip_list3):
```

```
6     print(item)
```

This is the above output but in a list form:

`list(zip(zip_list1,zip_list2,zip_list3))`

And here's zip with tuple unpacking:

```
1 for a,b,c in zip(zip_list1,zip_list2,zip_list3):
2     print(b)
```

In Operator

We can use this to check if something is in a list, strings, dictionaries. It will output a boolean True or False. Basic syntax is:

`thing_2_check in [ur_object_here]`

List: Is 2 in a list of 1,2,3? This will output True. If I swap 2 for 'x', it will output False.

`2 in [1,2,3]`

String: Is 'a' in 'a world'? This will output True.

`'a' in 'a world'`

Dictionary: I can check for keys or values. I setup a dictionary then need to add `.values()` or `.keys()`

```
1 d = {'k1':123}
2
3 123 in d.values()
```

Min/Max

There's a built in Min and Max function to check the min and max in a list. Syntax is `(min(list))`` and (max(list))```

Random Library

There's a library called random which has functions. These functions can generate random numbers. The basic syntax is:

`from random import function`

Select Random Functions:

1. Shuffle - Scrambles a list. So for example let's scramble a list and then output the new scrambled list:

```
1 mylist = [1,2,3,4]
2 from random import shuffle
3 shuffle(mylist)
4 mylist
```

2. Random number generator - Randint will generate a random number within a range. The basic syntax is below:

```
1 random import randint
2 randint(x,y)
```

Input Function

This let's you accept user input. The basic syntax is `input(ur_input_here)`. To save the input, you can cast it as a variable. So the example syntax would be `result = input(ur_input_here)`.

Result would be a string though. So you'd want to then cast the result as another type. So the example would be:

`result = int(input('Fav number:'))`

Temperature Converter Program

After I got to the end of the Udemy course and it was testing time, I tried to write a program that converts temperature into Celsius and Fahrenheit. I used this code as the starting point.

Pieces of code I'll need to understand the notes going forward:

1. Temperature Converter Baby Code.py - For base baby code
2. Temp Convert Phil 1.py - For Step 1 code
3. Temp Convert Phil 2.py - For Step 2 code
4. Temp Convert Phil 2 - Anson Mod Final.py - For Step 2 code - Final Code

My base baby code

So here's the basic code I made using the starting point. I added in the temperature rounding and the Unit checking.

```
# Intro Words
# This is a beginner program for Anson to learn how to make a simple
# temperature conversion program. This will convert from C to F and in reverse

temp = float(input("Enter temperature = "))
unit = input("Enter C or F (for Celsius or Farhenheit) = ")



if unit == "F" or unit == "f":
    fahrenheit = (temp * 9/5) + 32
    print(f'{round(fahrenheit,2)} F')
elif unit == "C" or unit == "c":
    celsius = (temp - 32) * 5/9
    print(f'{round(celsius,2)} C')
elif unit != "F" or unit != "f" or unit != "C" or unit != "c":
    print("Not a valid unit, try again.")
```

Next steps

1. I want to have the code check for errors. So if I put in temp = asf and unit = F, I want it to tell me I'm bad.
2. I want the code to loop through 3 times. So if I make mistakes, I want it to re-prompt me again to input my values. After 3 guesses, it quits out.

Step 1 Code

The base code is "Temperature Converter Baby Code.py". Then I wanted to add in mistake proofing so if I put in `asdf` as the temperature, it would say "Oh you messed up, try again" and reprompt me. Jwaz nerd chat introduced me to "try/except" which is explained nicely here. And they also introduced me to `NameError` and `ValueError` which is explained here.

Try/Except doesn't won't set anything if it doesn't work though. It only sets the variable if it works. So I'm getting a NameError with this code I came up with for Step 1 (I got the idea for checking if the value is a float from here):

```
# Intro Words
# This is a beginner program for Anson to learn how to make a simple
# temperature conversion program. This will convert from C to F and in reverse

temp = input("Enter temperature = ")
unit = input("Enter C or F (for Celsius or Farhenheit) = ")

try:
    temp_float = float(temp)
except ValueError:
    print("Not a valid temp")
 while temp_float == float:
    if unit == "F" or unit == "f":
        fahrenheit = (temp_float * 9/5) + 32
        print(f'{round(fahrenheit,2)} F')
    elif unit == "C" or unit == "c":
        celsius = (temp_float - 32) * 5/9
        print(f'{round(celsius,2)} C')
    elif unit != "F" or unit != "f" or unit != "C" or unit != "c":
        print("Not a valid unit. Try again.")
if temp_float != float:
    print("You messed up. Try again.")
```

Phil helped me with the code "Temp Convert Phil.py". His code looks like:

```
temp = (input("Enter temperature = "))
unit = input("Enter C or F (for Celsius or Farhenheit) = ")

try:
```

```
    temp_float = float(temp)
except ValueError:
    print("Not a valid temp")
    quit()

if not temp_float:
    print("You messed up. Try again.")
    quit()

if unit.lower() == "f":
    fahrenheit = (temp_float * 9/5) + 32
    print(f'{round(fahrenheit,2)} F')
elif unit.lower() == "c":
    celsius = (temp_float - 32) * 5/9
    print(f'{round(celsius,2)} C')
else:
    print("You messed up. Try again.")
```

Phil's notes on the code:

1. He taught me that `quit()` or `exit()` are things that exist. So now the code just quits out when I put in the wrong values.
2. A while loop I would reach for if I had more than one thing to iterate through, but in this situation we have 1 unique thing to make a decision on 1 time. I might use a while if I had 1000 temperatures but I couldn't be sure how many times I needed to go through the processing for conversion
3. The if elif else is the right way to think abou tthe linear decision process, if one thing, otherwise if another thing, otherwise here's what to do when you're out of options
4. Finally, because in my solution I didn't use a loop of any kind, pass doesn't work for this either. In a loop a pass would be proper if you needed a reason to exit the loop but continue to the logic after the loop. Instead, when you need to exit the whole program, exit() or quit() would be better tools

Step 2 Code

Now I want the code to repeat itself. I want the code to allow me 3 tries before quitting out. I had the ideal of using true or false booleans and while loops but wasn't sure how to bridge the gap. Phil helped with this code called "Temp Convert Phil 2.py":

```
# Instantiate initial true flag to enter loop
run_loop = True
```

```
# While set to go
while run_loop:
    # Try to capture a float at input time so we don't have to parse it later
    try:
        temp = (float(input("Enter temperature = ")))
    except Exception:
        print("Input isn't a temperature; try again")
        # If we can't establish a float for the first input, we'll simply skip the rest of this iteration and never set
        # the run_loop flag to false, allowing loop to continue
        # Exception catches all errors
        # more info here: https://docs.python.org/3/library/exceptions.html
        continue

    # Instantiate a sub loop
    run_loop_sub = True

    while run_loop_sub:
        try:
            unit = str(input("Enter C or F (for Celsius or Farhenheit) = "))
        except Exception:
            print("Input needs to be c/C or f/F")
            # Harder to hit this since "" is a string in input, but if it fails for whatever reason
            # just try again
            continue

        if unit.lower() == "f":
            fahrenheit = (temp * 9/5) + 32
            print(f'{round(fahrenheit,2)} C')
            # Completion condition met, set loop flag to false to exit loop after this iteration
            run_loop_sub = False

        elif unit.lower() == "c":
            celsius = (temp - 32) * 5/9
            print(f'{round(celsius,2)} F')
            # Completion condition met, set loop flag to false to exit loop after this iteration
            run_loop_sub= False

        else:
            print("You need to enter either c/C or f/F")
```

```
        # There is no satisfactory completion here, so don't set the close flag


    # If we make it here, that means that the sub while loop was satisfied, and there is no further exceptions to
    # skip this flag; we can probably end the loop
    run_loop = False
```

Notes on the code:

1. Instantiate to represent an instance
2. Try/Except has the `Exception` class which catches all built-in, non-system existing exceptions. This is what I wanted when I was messing with `ValueError` and `NameError` above.
3. Instead of the `!=` I was using to check for the case, the way Phil handled it was to just parse the input into a string then uses `unit.lower()` to set the case to lower case only. That way it doesn't matter what the case is.
4. Comments are good
5. Loops and subloops are a thing. Could have a overall loop and loops underneath the overall loop

So now I got most of the code working. "Temp Convert Phil 2.py" aka the code above will keep looping until it gets the correct answer. But now I want to add in the 3 strikes or the code ends. So my code is found in "Temp Convert Phil 2 - Anson Mod Final.py". It looks like:

```
# Instantiate initial true flag to enter loop
run_loop = True
# Set global variable to count
retry = 0
# Runs code while retry is under 3
while retry < 3:


    # While set to go
    while run_loop:
        # Try to capture a float at input time so we don't have to parse it later
        try:
            temp = (float(input("Enter temperature = ")))
        except Exception:
            # Exception catches all errors
            # more info here: https://docs.python.org/3/library/exceptions.html
            print("Input isn't a temperature; try again. Max 3 attempts. Attempts:",retry+1)
            retry += 1
```

```
        # If we can't establish a float for the first input, we'll simply skip the rest of this iteration and never set
        # the run_loop flag to false, allowing loop to continue
        # Print is setup so it tells me how many attempts I'm at and shows the count
        break
        # Ends the program if I guess too much

    # Instantiate a sub loop
    run_loop_sub = True

    while run_loop_sub:
        try:
            unit = str(input("Enter C or F (for Celsius or Farhenheit) = "))
        except Exception:
            print("Input needs to be c/C or f/F")
            # Harder to hit this since "" is a string in input, but if it fails for whatever reason
            # just try again
            continue

        if unit.lower() == "c":
            fahrenheit = (temp * 9/5) + 32
            print(f'{round(fahrenheit,2)} F')
            print('You know the temp now!')
            # Completion condition met, set loop flag to false to exit loop after this iteration
            run_loop_sub = False

        elif unit.lower() == "f":
            celsius = (temp - 32) * 5/9
            print(f'{round(celsius,2)} C')
            print('You know the temp now!')
            # Completion condition met, set loop flag to false to exit loop after this iteration
            run_loop_sub= False

        else:
            print("You need to enter either c/C or f/F")
            # There is no satisfactory completion here, so don't set the close flag

        # If we make it here, that means that the sub while loop was satisfied, and there is no further exceptions
to
        # skip this flag; we can probably end the loop
        run_loop = False
```

```
    #if retry == 3:
    #    run_loop = False
        # Not sure if this helps or hurts
        # After experimenting, it doesn't seem to matter if it's here
```

Notes on the code:

1. There's a hang up, the code doesnt close cleanly like it did in "Temp Convert Phil 2.py" after the temperature is input correct. But it does work.
2. When the 3 guesses are put in, so "asdf" for the temperature, I created a print that warns how many attmepts are left and what attempt number we're on.
3. The last 4 lines of code don't seem to do much. Not sure why I can just run a while loop as is.
4. Learned what a global variable is.
5. Learned what a counter is and how break works.

Notes - Tues 01/12/21

Tuesday 01/12/21

Methods and Python Documentation

Objects in Python have methods. So for example the `.split()` and `.append()` are types of methods. A list of the methods can be found by just typing in `mylist.` and seeing the popup window in VSCode. You can also use the help function which would be `help(mylist.mymethod)` which will output help. It's like `man` in the terminal.

For more in-depth documentation, you can go to the official Python documentation.

Intro to Functions

Functions allow us to create blocks of code that can be repeated or executed multiple times. This unlocks more capabilities and problem-solving.

def Keyword - Define a Function

Creating functions uses specific syntax. We use the `def` keyword to define a function. This includes the def keyword, indentation, and proper structure. So let's look at the basic syntax:

```
def name_of_function():
    ```
    Docstring
    ```
    then_your_code_goes_here
```

1. def = Tells Python you're creating a function
2. name_of_function = This is the name of your function. Note it uses "snake casing" which is the lower case words seperated with underscores _. This is the standard Python way to

note functions so it's easy to ID later.
3. parenthesis () = We can pass in arguements or parameters into the function. This sets up future potential.
4. colon : = Indicates to Python and upcoming indented block which will be what's inside the function.
5. Docstring ``` = It's a multiple format to put in comments about the function. I've been using it in Markdown to show lines of code and been using multiple lines of # to do the same thing.
6. Then you write your code that you want the function to execure.

- Note that functions can accept arguements to be passed by the user.

Typically we use `return` keyword to see the result of the function instead of printing it out. We can use `return` to assign the output of the function to a new variable. So this "saves" the output. Return will be covered in depth later.

```
# random method to show what they are
mylist = [1,2,3,4]
mylist.pop()
4
```

```
# help function for methods
help(mylist.pop)


Help on built-in function pop:


pop(index=-1, /) method of builtins.list instance
    Remove and return item at index (default last).


    Raises IndexError if list is empty or index is out of range.
```

Intro to Functions

Functions allow us to create blocks of code that can be repeated or executed multiple times. This unlocks more capabilities and problem-solving.

def Keyword - Define a Function

Creating functions uses specific syntax. We use the `def` keyword to define a function. This includes the def keyword, indentation, and proper structure. So let's look at the basic syntax:

```
def name_of_function():
    ```
    Docstring
    ```
    then_your_code_goes_here
```

1. def = Tells Python you're creating a function
2. name_of_function = This is the name of your function. Note it uses "snake casing" which is the lower case words seperated with underscores _. This is the standard Python way to note functions so it's easy to ID later.
3. parenthesis () = We can pass in arguements or parameters into the function. This sets up future potential.
4. colon : = Indicates to Python and upcoming indented block which will be what's inside the function.
5. Docstring ``` = It's a multiple format to put in comments about the function. I've been using it in Markdown to show lines of code and been using multiple lines of # to do the same thing.
6. Then you write your code that you want the function to execure.

- Note that functions can accept arguements to be passed by the user.

Typically we use `return` keyword to see the result of the function instead of printing it out. We can use `return` to assign the output of the function to a new variable. So this "saves" the output. Return will be covered in depth later.

```
# Let's create a function where we're adding numbers


def add_function(num1,num2):
    return num1+num2


# Now with return, we're able to save the output of the function as the "result"
# We want add_function to run with the numbers 1 and 2
result = add_function(1,2)


# Now we can see what the result from above
print(result)
3
```

Basic Functions in Python Practice

Let's create a function called `say_hello` which will output Hello and then a name. In this example I define the function as ```say_hello``. Then on the next indented line, I tell python what the function does. Then run the function to get the output.

```
def say_hello():
    print('smell')
    print('you later')

say_hello()

smell
you later
```

Function Variable

Now let's add in a parameter variable which we can pass using an f-string literal. So we'd add 'name into the parathesis after `say_hello` and then the next line has an f-string literal which contains the variable. Then we want to output the function but add in a name. Note that the variable 'name' could be anything but we want it to be easily understood. For example I used this to show how many tries were left in my Temperature Converter Program.

Basic Syntax would be `def ur_function(ur_variable)`

```
def say_hello(name):
    print(f'Hello {name}')

say_hello('Girlshark')

Hello Girlshark
```

Note that if I run `say_hello()`, I'll get a "positional arguement error" which means I missed putting something in for the name.

```
say_hello()



TypeError                                 Traceback (most recent call last)
<ipython-input-32-faa5fc24272a> in <module>
----> 1 say_hello()

TypeError: say_hello() missing 1 required positional argument: 'name'
```

Default Value

Let's say that we don't want that error to happen. We can setup something called a "default value". So this is the value that would show up if we missed inputting something after `say_hello`. But then it'd work normally after we put in an input.

Basic Syntax would be `def ur_function(ur_variable='default_value')`

```
def say_hello(name='stupid; You forgot to put something in'):
    print(f'Hello {name}')


say_hello()


Hello stupid, You forgot to put something in
```

```
say_hello('wizbro')


Hello wizbro
```

Return Keyword

As mentioned above, we typically don't have functions just print stuff out. Otherwise we'd just print them out instead of setting up a function. Instead, we want to use the `return` keyword. `print` just shows the output while `return` allows you to save the output as a variable.

Note in the below example, we've set 2 variables in the define function (aka first line) line.

Basic Syntax would be (note there's no parenthesis when using return):

```
def ur_function(ur_variables):
    return do_something_to_variables
```

```
def add_num(num1,num2):
    return num1+num2


# See I saved it
result = add_num(10,20)
result


30
```

Let's compare what happens when we try to save the output from `print` vs `return`. Note that we can use these together. It just yields a different output.

```
# Let's compare print vs return
def print_result(a,b):
    print(a+b)

def return_result(a,b):
    return a+b

# I tried to save the result of print_result then check the type.
# Note that the result is nonetype

result = print_result(10,20)
type(result)

30
NoneType
```

```
# Now compare that the the return function
# I saved the return result then output it

result2 = return_result(10,20)
result2

30
```

```
# Then I checked the type which gives me integer
type(result2)

int
```

```
# Combine print + result
# This isn't 100% needed. Good for things like troubleshooting.

def myfunc(a,b):
    print(a+b)
    return a+b

result = myfunc(665,1)
```

```
666
```

Note that python is dynamically typed so you don't have to tell Python what data/object type you're inputting before you run the code. So let's say you've got user input. That `input` value yields a string and you might want a number instead.

Functions with Logic

Reminder that logic is stuff like if or else statements.

For example, let's write a function that checks if a number is even.

```
def even_check(number):
    result = number % 2 == 0
    return result

even_check(20)

True

even_check(21)

False
```

Above has the noob version of the definition. But you can compress the return line if there's a boolean check.

So the above result is: `result = number % 2 == 0`

And the advanced mode is: `return number % 2 == 0`

```
def even_check(number):
    return number % 2 == 0

even_check(16)

True
```

Now let's check a list. We want to return True if any number is even inside the list.

```
def check_even_list(num_list):
    for number in num_list:
    # check all the variable objects in the defined list
        if number % 2 == 0:
        # if any numbers have a remainder of 0 after dividing by 2
            return True
        # return the True boolean
        else:
            pass
        # otherwise pass or end the code

check_even_list([1,2,3])


True
```

```
check_even_list([3,5,7])
# If I run this list, I currently only output if there's something even. So there's no output. That's what the pass does.


def check_even_list(num_list):
    for number in num_list:
    # check all the variable objects in the defined list
        if number % 2 == 0:
        # if any numbers have a remainder of 0 after dividing by 2
            return True
        # return the True boolean
        else:
            pass
        # otherwise pass or end the code
```

Now let's have an output that when there's a non-even number, we get False. If we just modify `pass` in the last night to `return False`, the code will only run once since we're essentially telling the code to return True and False at the same time. Instead, we need to setup the code so that there's multiple loops. First is the even number = output True loop. Then we setup a second loop to check for odd numbers. So the new return False line would align with the For loop.

```
def check_even_list(num_list):
    for number in num_list:
        if number % 2 == 0:
            return True
```

```
        else:
            pass
# Above the code is what we had before.

    return False
# So now what this is saying, run the first loop to check even number = True.
# If that loop doesn't work, the code is stopped with pass and
# then it will return False

check_even_list([1,2,4])

True

check_even_list([1,3,5])

False
```

Ok now let's say we want the program to return all the even numbers in a list. Otherwise it will return an empty value. Note you can setup a placeholder function for variables you're not sure of yet.

```
def check_even_list(num_list):
    even_numbers = []
    # Here is the empty placeholder list

    for number in num_list:
        if number % 2 == 0:
            even_numbers.append(number)
            # Here we're editing the even_number variable using append
            # Remember append adds new objects to the end of the list
            # so we're setting a variable and checking that variable to be even
            # then we're adding those numbers to the list

        else:
            pass
            # if this doesn't work out, it'll just stop the program

    return even_numbers
    # here we're returning the even_numbers variable
```

```
check_even_list(range(0,11))
# finally we're running the function with the range 0-10


[0, 2, 4, 6, 8, 10]


check_even_list([1,3,5,7])
# running another list with no even numbers and we get an empty list


[]
```

Functions and Tuple Unpacking

Now we're going to return mulitiple items using tuple unpacking which was discussed previously. Let's run an example to remind me how Tuple Unpacking works.

```
stock_prices = [('APPL',200),('GOOG',400),('MSFT',100)]


for item in stock_prices:
    print(item)


('APPL', 200)
('GOOG', 400)
('MSFT', 100)


for ticker,price in stock_prices:
    print(price-(.15*price))
    # we're unpacking the price but also
    # showing the price with a 15% loss - F


170.0
340.0
85.0
```

Turns out we can use tuple unpacking for a function too. Let's setup a new example where we have a list of work hours and we want to find the employee of the year.

```
work_hours = [("Krabs",32),("Squidward",40),("Spongebob",2000)]
# Here's a list of the Krusty Krab employees and their hours
```

```
def employee_check(work_hours):
# we're setting up the function and variable

    current_max = 0
    employee_of_month = ""
    # This sets the current max variable as a placeholder for the hours later
    # and the employee of the month variable as a placeholder

    for employee,hours in work_hours:
        if hours > current_max:
        # this checks the hours from the tuple
        # if it's more the current_max hours
        # it resets the variables current_max and employee_of_the_month
            current_max = hours
            employee_of_month = employee
        else:
            pass
        # if the above loop is finished, then the code stops

    return (employee_of_month,current_max)
    # this returns the variables

result = employee_check(work_hours)
result
# this stores the function output as a new variable
# the current employee_check is now Spongebob

('Spongebob', 2000)

# We can do tuple unpacking on the function now
name,hours = employee_check(work_hours)
hours

2000

result

('Spongebob',2000)
```

Interactions Between Functions

Typically a Python script, code, program will contain several functions interacting with each other. We'll take the results of one function and use them as inputs to another function. To show this, we're going to write a 3 cup monte shuffle game. We won't see the graphics but mimic the effect with lists. And we're not going to see the shuffle. The user will randomly guess.

First, let's have a reminder that the `random` library exists. That library has a `shuffle` function which will randomy shuffle a list of numbers. So let's look at that.

Note that a function will be needed to store the `shuffle` output.

```
from random import shuffle
# this imports shuffle from the random library

example_list = [1,2,3,4,5,6,7]
# this generates a list from 0-7 called example_list

shuffle(example_list)
# note this can't be stored and can't be returned as is

example_list

[1, 3, 6, 4, 7, 5, 2]

def shuffle_list(mylist):
    shuffle(mylist)
    return mylist
# this function stores the shuffle so I can use it repeatedly

result = shuffle_list(example_list)
# variable result uses function shuffle_list to shuffle the example_list i had above
result

[2, 3, 1, 4, 5, 7, 6]

# Now we can simulate O as the ball in the cup shuffle game
# let's setup a list then use the shuffle_list function from above
mylist = ['','O','']
```

```
shuffle_list(mylist)
```

```
['O', '', '']
```

```
# Now let's setup the player's guess
def player_guess():

    guess=''

    while guess not in ['0','1','2']:
        guess = input("Pick a location: 0, 1, 2")

    return int(guess)

player_guess()
```

```
2
```

```
# Now let's write a function to check the guess

def check_guess(mylist,guess):
    if mylist[guess] == "O":
        print("Correct!")

    else:
        print("Wrong guess!")
        print(mylist)

# Now let's combine it all

# Initial List
mylist = ['','O','']

# Shuffle List
mixedup_list = shuffle_list(mylist)

# User Guess
guess = player_guess()
```

```
# Check Guess
check_guess(mixedup_list,guess)


Wrong guess!
['O', '', '']
```

Overview of Quick Function Exercises #1-10 (aka notes before the Quiz coming up)

Okay so you now know functions!

A big part of this section of the course will be testing your new skills with exercises. We have 3 main parts of exercises.

Part 1: 10 In Course Coding Exercises

We're going to start off with just the basics with a series of 10 problems. These problems should feel relatively easy, just some quick exercises to get you comfortable with the syntax of functions. If you feel uncomfortable with these, check out lecture 26 for some useful links for warm-up problems from codingbat.com , but hopefully these exercises should feel relatively easy.

These are in-course coding exercises. Solutions can be found linked here:

https://docs.google.com/document/d/181AMuP-V5VnSorl_q7p6BYd8mwXWBnsZY_sSPA8trfc/edit?usp=sharing

In between these in-course coding exercises we'll have a quick lecture on *args and **kwargs.

Part 2: Function Practice Exercises

Here we'll have a jupyter notebook with some exercises for you to answer, we'll have a quick overview lecture, and then have you attempt problems, afterwards we'll have an explanatory solutions video. These problems are ranked WarmUp, Level 1, Level 2, and Challenge. You should feel comfortable with Warmup and Level 1 and Level 2. Challenge problems here are very difficult, so don't feel bad if you don't want to attempt them yet! :)

After this we'll cover a few more topics through some videos.

Part 3: Function and Methods Homework

We finish off this section with even more exercises! Here we have various function word problems for you to solve, again in a notebook and we will cover the solutions in a video afterwards.

Best of luck! If you have any questions, post to the QA forums and we'll be happy to help you out!

Notes - Friday 01/15/21

Friday 01/15/21

Today I wanted to work on the test and comprehension part of the course which can be seen here. They threw in something called *args and **kwargs in the middle of my coding practice. So let's learn about args.

*args and **kwargs

*args

These are functional parameters. Arguments are `*args` and pronounced "star args" and keyword arguments are `**kwargs` and pronounced "double star qw-ar-gs". These are things in Python functions that allow for the code to accept an abitrary number of arguments and keyword arguments without pre-defining parameters in the function calls.

So let's say we want to setup a function that will return 5% of the sum of 2 numbers. But what if I then want to change the function to take a variable or arbitrary amount of arguments. So then I don't have to define the number numbers the function will do. Note that the `*args` will become a tuple. The `*args` could technically be anything; it could be `*urwordshere` or `*spam`. Best practice is to call it `*args`.

CBAkwarg01.png

**kwargs

Python offers a way to handle an arbitrary number of key word arguements. It creates a dictionary of key value pairs. So it does the same thing that `*args` does where that returns a tuple, but instead returns a dictionary. Then the user can define values in the dictionary that can be messed with inside the function. Again `**kwargs` could be anything after ``**urnamehere`. Best practice is to call it` **kwargs``` so it's easy to recognize.

Why *args and **kwargs?

They're useful to use when you pull in outside libraries. They might not be useful now but will be useful later. Note that they can be combined

CBAkwarg02.png

Things I learned from the Test - Skyline question

The question being asked was:

"Define a function called **myfunc** that takes in a string, and returns a matching strnig where every even letter is uppercase and every odd letter is lower case. Assume that the incoming string only contains letters and don't worry about the numbers, spaces, or punctuation. The output string can start with either upper or lower case. The letters should alternate throughout the string. Just provide the definition for the function. You don't need to run it. Remember you need to use the *return* function and not print."

The answer was:

```
def myfunc(x):
    out = []
    for i in range(len(x)):
        if i%2==0:
            out.append(x[i].lower())
        else:
            out.append(x[i].upper())
    return ''.join(out)
```

Things I learned:

- I didn't realize that you could combine range and length to count the number of letters.
- .join - Joins all items in a tuple into a string with a hash character as a seperator
- Remember that .append is what modifies the list that variable out calls out
- An explanation on what is going on in this code can be found here

Notes - Sunday 01/17/21

TIL

Today I'm working on some practice problems. I'm going to take some notes on what I learned while I go through these problems. Unfortunately the solutions include things we've never gone over in the lectures. Reference this link for the questions and this link for the answers.

- Remember to read over the notes carefully and don't rush to solve a problem for no reason.

- If you're using string.split() function, remember that the resulting list will have index positions based on the entire string. Then you can breakdown the string further. For example:
  `my_list_of_words = some_words.split() = ['Hello World']` (this isn't perfect syntax, just go with it) yields `['Hello','World']`
  This means that `my_list_of_words[0]` yields `'Hello'` and `my_list_of_words[1]` yields `'World'`
  Then you can break this down further to get just individual letters. For example `my_list_of_words[0][0]` yields `'H'` and `my_list_of_words[1][0]` yields `'W'`

- `.join()` method: So you can use this method to combine or join strings. The basic syntax is: `"stuff_between_ur_words".join(ur_variable)`
  So let's look at a practice problem to explain it:

  ```
  MASTER YODA: Given a sentence, return a sentence with the words reversed


  master_yoda('I am home') --> 'home am I'
  master_yoda('We are ready') --> 'ready are We'
  ```

CBAPractice01.png

TIL Continued...

- You can find the absolute value of a number by using abs. The syntax would be `abs(ur_number)`

- Assignment operators exist. Here's a list of them. Things like += and = can help. I didn't know they existed.

- Remember that you can get the sum of numbers using `sum(ur_numbers)`

- When you have a loop, order the loop matters. It will literally execute the logic line by line so if you're not setting up the loop in the order you want it to check in, it won't output what you want.