Let's get started.

The Basics

Python has eight basic data types:

- Integer [int] - whole numbers: 3, 300, 26, etc.
- Floating Point [float] - numbers with a decimal point: 2.3, 4.6, 100.0.
- String [str] - Ordered sequence of characters: "hello", "Sammy", "2000", "コンピュータ"
- List [list] - Ordered sequence of objects: [10, "hello", 200.3]
- Dictionary [dict] - Unordered Key:Value pairs: {"mykey":"value,"name":"Frankie"}
- Tuple [tup] - Ordered immutable sequence of objects: (10, "hello", 200.3)
- Set [set] - Unordered collection of unique objects: {"a","b"}
- Boolean [bool] - Logical value indicating *True* or *False*

Handling Numbers:

You can perform basic math in Python.
Use `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division.

```
2+1
#####
2-1
#####
2*2
#####
3/2
```

Note that nothing is being output by these basic lines of code, as nothing is being printed. More on that later.

Modulo [mod] operations return the remainder after division. Use `%` to divide and return remainder.

```
7%4
#####
#This would return '3'. However nothing will be printed to the console. See
above.
```

Exponents are expressed with two asterisks followed by the exponent number. For example, 2^3 can be expressed as `2**3`

Floor division is expressed with two forward slashes. This removes any remainder, rounded down.

```
7//4
#####
#This would return '1', because 4 does not divide evenly into 7 more than
once.
```

Order of Operations can be performed with parentheses.

```
(2+10)*(10+3)
#Will return a different result than
2+10*10+3
```

Variables

Variables are names assigned to data types for easy categorization. For example, if I set `my_cats = 5` then any time `my_cats` is referenced in the code, it will always equal 5.
Variable names cannot start with a number. There cannot be any special characters in a variable name.
Best practice is that variable names are all lowercase (aside from global variables). Avoid words that are already special values in Python, such as "list" and "str".

Python uses dynamic typing, which means that you can reassign values to different data types. For example:

```
my_cats = 5
my_cats = ["Goose","Uno"]
```

This may cause bugs in your code if you aren't careful, so be vigilant in your code, and use `type()` to find the data type of a variable if you are confused (for example `type(my_cats)`)

A variable can reference itself upon reassignment. For example:

```
a = 10
a = a + a
```

Now every time `a` is called, it will add 10 to its value.

Strings

Strings are sequences of characters using the syntax of single or double quotes.
`'hello'`, `"Hello"`, `"I don't do that"`

Indexing and Slicing can be used to grab subsections of a string. Indexing syntax uses brackets `[]` and is used to grab a single character from the string. *Indexing values always start at 0.*

String	H	E	L	L	O
Index Position	0	1	2	3	4
Reverse Index	0	-4	-3	-2	-1

For example, if we just wanted to grab the letter E from the string "HELLO", we would use
`"HELLO"[1]`
Reverse indexing can be used in place of standard indexing if you want to work backwards from the string. For example, if we wanted to grab the letter E from "HELLO" using reverse indexing, we would use
`"HELLO"[-4]`
Slicing lets you grab a subsection of multiple characters. It also uses square brackets, and uses the syntax of `[start:stop:step]` where `start` is a numerical index for the start of the slice, `stop` is the index the selection goes up to *but does not include,* and `step` is the size of the jump taken between slices.

Spaces do count as characters within a string, and therefore have their own index values. They are one of a few characters often referred to as whitespace by the development community.

To **print** a string, use the print function with proper syntax.

```
print("Hello World!")
#####
#OUTPUT > Hello World!
```

Escape Sequences `\n` will print everything following the escape sequence in a new line. Note that this does not work if a string is being returned from a function.
`\t` will add a tab (four spaces) following the command.

```
print("Hello World! \n I am learning escape sequences!")
#####
```

```
#OUTPUT > Hello World!
#I am learning escape seqences!
```

You can check the length of a string by using the `len()` function.

```
len('hello')
#####
#This will return '5', but note that it will not be printed to the console as
we did not use the print function.
```

Indexing syntax is as follows:

```
mystring = "Hello World"
mystring[0]
#####
#OUTPUT > H
```

where indexing a variable requires the variable name, followed by the square brackets, and the character you want indexed inside the brackets.

Slicing syntax is similar, where you have the variable name and square brackets. However, slicing returns a larger subsection of the string, and is expressed with colons. A basic string slice would consist of starting in the middle of the string and returning the rest until the end.

```
mystring = 'abcdefghijk'
mystring[2:]
#####
#OUTPUT > cdefghijk
```

Similarly, starting at the beginning of the string and ending toward the middle requires a colon in the first position of the bracket.

```
mystring = 'abcdefghijk'
mystring[:3]
#####
#OUTPUT > abc
```

As mentioned before, the stop index (second position in a slice) does not include the selected character. In the example above, "3" technically indicates the letter 'd', but it is not included in the resulting slice.

The third and final position in a slice is step size. It indicates how big of a step to take in the slice. For example, `mystring[::2]` would return every other letter, or every second letter.

```
mystring = 'abcdefghijk'
mystring[::2]
#####
#OUTPUT > acegik
```

Combining all of these principles, slices can have a start, stop, and step size all at once.

```
mystring = 'abcdefghijk'
mystring[2:7:2]
#####
#OUTPUT > ceg
```

Step sizes can also be reversed, like indexes. This can also be used to reverse a string.

```
mystring = 'abcdefghijk'
mystring[::-1]
#####
#OUTPUT > kjihgfedcba
```

Strings are **Immutable**, you cannot index/slice part of a string and set it as its own object/variable. However, you can reference an indexed variable within a new variable. For example, the following is **NOT** possible in Python:

```
name = "Sam"
name[0] = "P"
#In this example, we are attempting to reassign the first letter of our 'name'
string from 'S' to 'P', which is not possible due to the immutable nature of
strings.
```

However, the following **IS** possible:

```
name = "Sam"
lastname = name[1:]
#In this case, we are creating a new variable by referencing a slice of
another string variable. our 'lastname' variable will end up being "am".
```

You can **Concatenate** strings with `+`

```
name = "Sam"
lastname = name[1:]
print('P'+lastname)
#####
#OUTPUT > Pam
```

You can also concatenate via multiplication with `*`

```
letter = 'z'
print(letter*10)
#####
#OUTPUT > zzzzzzzzzz
```

One thing worth noting is that 2+3=5, but '2'+'3'='23'

String Formatting

Methods are a list of actions that can be performed within a string. For example, let's say `x = "hello world"`. If you were to type `x` and press [tab], a list of methods will pop up. There are several methods. Here are a few notable ones:

- `upper()` will capitalize everything in a string
- `lower()` will lowercase everything in a string
- `split()` will create a list from the string. `["hello","world"]`. This split is based on whitespace if left blank, or the letter that's passed in. for example, `x.split('l')` will output `['he', '', 'o wor', 'd']`. Note how the letter that's passed in gets omitted from the split.

Print Formatting allows you to use `.format` to insert arguments into strings. The syntax looks like this:

```
print('This is a string {}'.format('INSERTED'))
#####
#OUTPUT > This is a string INSERTED
```

Where the curly brackets define where an inserted argument will go, and the second set of parentheses define what will go there.
As another example:

```
print('the {} {} {}'.format('fox','brown','quick'))
#####
```

```
#OUTPUT > the fox brown quick
```

Each argument was presented as a list within the second set of parenthesis, and printed in the order supplied.
However, the presented variables can be indexed. Each argument is one index number. The indexes go in the curly brackets.

```
print('the {2} {1} {0}'.format('fox','brown','quick'))
#####
#OUTPUT > the quick brown fox
```

You can also set keywords for your inserted values. This is like assigning arguments as variables.

```
print('the {q} {b} {f}'.format(f='fox',b='brown',q='quick'))
#####
#OUTPUT > the quick brown fox
```

Float Formatting is used to increase the precision of a float. For example:

```
result = 100/777
print(result)
#####
#OUTPUT > 0.1287001287001287
```

If I wanted to truncate that number in print formatting, I'd use this syntax:
`{value:width.precision f}`
Where:

- value = our inserted variable
- width = minimum amount of characters to be used, including decimal point and whitespace
- precision = number of decimal places to print (following the same rules as slicing.)
  So using this formatting, we can truncate this number in print:

```
result = 100/777
print('{r:1.3f}'.format(r=result))
```

This also works with whole numbers - the precision modifier will not affect the whole number, but will still modify anything to the right of the decimal point.

F-Strings provide a similar outcome with different inputs. An f-string will allow you to insert a pre-defined variable directly into a string without having to format it. For example:

```
name = 'Author'
print(f'Hello, my name is {name}')
#####
#OUTPUT > Hello, my name is Author
```

Lists

Lists are ordered sequences that can hold several object types. They use brackets and commas to separate the objects. They support indexing, slicing, and can be nested.
You can check the length of a list with the `len()` function. Lists can be indexed and sliced like strings, where index numbers represent objects in the list rather than individual characters. For example, let's say `my_list = ["one","two","three"]`

List Objects	"one"	"two"	"three"
Index Position	0	1	2

If you were to index this list, it might look like this:

```
my_list = ["one","two","three"]
my_list[1]
#####
#OUTPUT > two
```

If you concatenate two lists, it joins them together as a single list.
Lists are *not* immutable, meaning a single index can be redefined.

```
my_list = [1,2,3]
my_list[0] = "ONE"
#####
#Now 'my_list' is ["ONE",2,3]
```

You can add data to a list with the `.append()` method. This will add whatever is passed into the method to the end of the list.

```
my_list = [1,2,3]
my_list.append(4)
```

```
#####
#Now 'my_list' consists of [1,2,3,4]
```

Conversely, you can remove an item from the end of a list with the `.pop()` method. This will also return your removed value. You can remove an item from anywhere in the list by passing in an index position. For example:

```
my_list = [1,2,3,4]
my_list.pop(1)
#####
#Now 'my_list' consists of [1,3,4]
```

You can sort a list alphabetically or numerically with the `.sort()` method. This is a NoneType, so it will not return a value. However, next time you call your sorted list, it will be indexed in order. It will not do this if you have a mix of strings and integers. Running `.reverse()` will reverse your list. This is also a NoneType.

Dictionaries

Dictionaries are unordered mappings for storing objects. Rather than being in an ordered sequence, dictionaries use key:value pairing. This allows you to quickly grab objects without needing to know an index location. Dictionaries use curly brackets and colons to signify keys and associated values.
`{'key1':'value1','key2':'value2','key3':'value3'}`

Dictionaries vs Lists
Dictionaries have objects retrieved by key name. Objects in dictionaries are unordered and cannot be sorted. Good for retrieving data whose index position isn't known or needed. Lists are ordered, sortable, and can be indexed and sliced. Essentially: easy data retrieval OR sortable and indexable data. In a dictionary, the keys are always strings.

Let's define the following dictionary:
`my_dict = {'key1':'value1','key2':'value2'}`
In order to retrieve any of the values, I just have to pass it into square brackets following the variable.

```
my_dict = {'key1':'value1','key2':'value2'}
my_dict['key1']
#####
#OUTPUT > value1
```

Dictionaries can contain various data types including lists and even other dictionaries.

Dictionary calls can be stacked. Let's say we have a dictionary that contains a list:
`dict = {'k1':[0,1,2]}`
If I wanted to call the number 1, I call the key, and then the index of the list. Both in separate square brackets.

```
dict = {'k1':[0,1,2]}
dict['k1'][1]
#####
#OUTPUT > 1
```

If you want to add a dictionary entry to the end of the dictionary, call the variable name, pass the new key into square brackets, and assign the new value like you would a variable name.

```
dict = {'k1':[0,1,2]}
dict['k2'] = 25
#####
#Now 'dict' contains two keys and values. {'k1':[0,1,2],'k2':25}
```

You can also use this method to overwrite existing dictionary entries.

Useful dictionary methods:

- `.keys()` - returns only the keys of a dictionary
- `.values()` - returns only the values of a dictionary
- `.items()` - returns key/value pairings in individual parentheses (tuples)

Tuples

Tuples are similar to lists, but they are immutable. Elements inside tuples cannot be reassigned.
Tuples use parentheses `(1,2,3)`
Tuples can hold multiple data types, and be indexed and sliced.
Some useful methods for Tuples:

- `.count(<data>)` - counts the occurrences of the passed-in data.
- `.index(<data>)` - indexes the FIRST occurrence of the passed-in data.
  For example:
  `t = ('a','b','a')`
  If we wanted to count how many instances of the string `'a'` were in this tuple, we would run
  `t.count('a')`
  which would return `2`

To index the *first* occurrence of `'a'`,
`t.index('a') which will return 0`

Since Tuples are immutable, data inside of them cannot be changed or reassigned. Good for data integrity; values cannot be accidentally reassigned.

Sets

Sets are unordered collections of unique elements. There can only be one representative of the same object. Create a set by defining a variable:
`myset = set()`
Add data to the set by using the `.add()` method.
`myset.add(1)`
Now calling `myset` returns `{1}`. Note that just because it's in curly braces, that doesn't mean it's a dictionary. You can also pass in a list with `set(<list>)` in order to only cast unique values.

Booleans

Booleans are operators that allow you to convey True or False statements. True & False must be capitalized when defining booleans, otherwise python will mistake it for a variable. None data types can be assigned as placeholders, but they must also be capitalized.

Basic File I/O

You can open a .txt file saved locally on your device with `open('filepath')`
Two important things to note here:

1. When you input the filepath, use double backslashes in windows. In MacOS and Linux, single forward slashes are fine.
2. The filepath MUST be a string in either single or double quotes.
   Let's set this function to a variable called `myfile`.
   If you want to call the contents of the .txt file, you'd use the `.read()` method, which looks like:
   `myfile.read()`
   This line of code will print the content of your file using python formatting (such as escape characters at line breaks, etc.) If you attempt to run the above code again, it would just output `''` - this is because there is an invisible 'cursor' that starts at the beginning of the .txt file, but it's at the end of the file when the `.read()` method is called. You can reset it with the `.seek(0)` method.

If you want your .txt file to display accurate line breaks, use the `.readlines()` method. This will still output `\n` at each line break, but it will be individual lines at the breaks rather than one

long string. Each line is defined as values in a list, and can be indexed as such.

If you use the `.open()` method, you also have to use `.close()` for the same file, or else it will be treated as constantly in use by your OS. You can keep from having to do this by using `with` and `as`.

```
with open('my\\filepath\\document.txt') as my_new_file
    contents = my_new_file.rend
```

Where `my_new_file` is a new variable name set for the text file. Notice the indent on the second line, the code in the second line only affects `my_new_file`. From there, we assigned a new variable, `contents` to read out the text file from the first line of code. This also keeps you from having to use `.seek(0)` every time you want to output the contents of the .txt file.

You can specify modes in the above code block in the parentheses of `open('my\\filepath\\document.txt*')`. You can specify modes where the asterisk is, preceded by a comma.

- `mode="r"` - allows your code to read only
- `mode="w"` - allows your code to write only, rendering an error from the `.rend()` method. Will create a file if none exist.
- `mode="a"` - allows your code to add to the end of your .txt file.
- `mode="rt"` - allows reading and writing
- `mode="wt"` - allows writing ands reading (overwrites current file or creates a new one if none exist)
  Extra notes about modes:
  `mode="a"` will append wherever your cursor happens to be. To properly use the "a" mode, use the `.write()` method rather than `.read()`.

Logical Operators

Comparison Operators allow you to compare variables and output a boolean value.

- `==` = "Are equal"
- `!=` = "Are NOT equal"
- `>` = "Is greater than"
- `<` = "Is less than"
- `>=` = "Is greater than or equal to"
- `<=` = "Is less than or equal to"

You can use these in code by writing `var1 == var2` where `==` is any of the above operators. Important: Capitalization matters! `Hi` is NOT equal to `hi`.

Strings and integers are different. `'2'` is not the same as `2`.

Logical Operators can be used to chain together comparisons. The keywords for these are `and`, `or`, and `not`.

You can run multiple simple comparisons at once, such as `1<2<3`. Alternatively you can write `(1<2) and (2<3)`

The logical operator keywords can be used to produce a single boolean output based on multiple comparison inputs.

- `and` - both comparisons are equally correct
- `or` - one of the comparisons is true
- `not` is used a bit differently. It returns the opposite boolean output. Syntax is `not(1==1)`. This example would return `False`.

If, Elif, Else Statements

These are **Control Flow Statements** and only execute code when certain conditions are met. Control flow statements use colons and indentation. Proper syntax is as follows:

```
if some_condition:
    #execute some code
```

Note the colon at the end of the statement, and the indentation on the second line. Going further, there are else statements, which is code that is run when the `if` condition is not met.

```
if some_condition:
    #execute some code
else:
    #do something else
```

Note that the `else` state does not have a condition. It will run in the event that the condition of the `if` statement is not met.
`elif` statements are a secondary condition to check for.

```
if some_condition:
    #execute some code
elif another_condition:
    #do something different
else:
    #do something else
```

For Loops

Many Python objects are iterable - you can iterate over every element in said object, such as elements in lists or characters in strings. For loops execute a block of code for every iteration.

For Loop Syntax:

```
my_iterable = [1,2,3]
for item_name in my_iterable:
    print(item_name)
#####
#OUTPUT > 1
#2
#3
```

This block of code does a few things. It first assigns a variable as a set. Then, the For Loop terms are defined. *For every item name in the defined variable, do this thing.* Then instructions are defined. Note the whitespace. The instructions defined in this example has the code print each item name, but you can set those instructions to anything.
The variable `item_name` was not previously defined in the code. When you begin a for loop with a known variable (such as `my_iterable` in this case) then you may define its contents with any name you'd like.

This is where control flow begins. You can chain for loops and other processes to begin manipulating data in a more complex way. For example

```
my_list = [1,2,3,4,5,6,7,8,9,10]
for num in my_list:
    if num % 2 == 0:
        print(num)
#####
#OUTPUT > 2
#4
#6
#8
#10
```

You can also us for loops on strings.

```
mystring = "Hello World"
for letter in mystring:
    print(letter)
#####
```

```
#OUTPUT > H
#e
#l
#l
#o
#
#W
#o
#r
#l
#d
```

Tuple Unpacking is a way to process data in Python that involves removing data from tuples into their own individual data types. For example, let's define a list of tuples:
`mylist = [(1,2),(3,4),(5,6),(7,8)]`
While there are technically 8 unique integers in this list, running `len(mylist)` would only return `4`, as there are only 4 tuples, each containing their own data. If you wanted to unpack these tuples, you might do this:

```
mylist = [(1,2),(3,4),(5,6),(7,8)]
for (a,b) in mylist:
    print(a)
    print(b)
#####
#OUTPUT > 1
#2
#3
#4
#5
#6
#7
#8
```

Note that the parentheses on `(a,b)` when defining this variable aren't strictly necessary. You can also only print one of the two members of the tuple by defining `print(a)` OR `print(b)`, but not both.

iterating through dictionaries is a unique challenge. Let's define an example dictionary:
`d = {'k1':1,'k2':2,'k3':3}`
If we were to try to print each item in the dictionary using a for loop, it would only print the keys and not the values.

```
d = {'k1':1,'k2':2,'k3':3}
for item in d:
```

```
    print(item)
#####
#OUTPUT > k1
#k2
#k3
```

To iterate the values along with the keys, you will need to use the `.items()` method.\

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

As you can see, this lists each key:value pair as tuples on their own line.
This means that you can use tuple unpacking to also unpack a dictionary:

```
d = {'k1':1,'k2':2,,'k3':3}
for key,value in d.items():
    print(value)
#####
#OUTPUT > 1
#2
#3
```

A **Nested Loop** is essentially a loop inside of another loop.

```
mylist = []

for x in [2,4,6]:
    for y in [100,200,300]:
        mylist.append(x*y)
#####
#OUTPUT > [200, 400, 600, 400, 800, 1200, 600, 1200, 1800]
```

While Loops

While Loops will continue to execute a block of code while a condition remains true. Syntax is as follows:

```
while some_boolean_value:
    #do something
```

where `some_boolean_value` is usually `x==true`. Note the indent on `#do something` - this is where you would put the block of code to be executed.
You can also append while loops with an else statement.

```
while some_boolean_value:
    #do something
else:
    #do something else
```

Useful & Important Keywords For While Loops:

- `break`: breaks out of the current closest enclosing loop.
- `continue`: goes to the top of the closest enclosing loop
- `pass`: does nothing at all.

`pass` is used as a placeholder for loops when you haven't written their instructions yet. For example:

```
x = [1,2,3]
for num in x:
    pass
```

This will do nothing, as the for loop is being passed.

`continue` is used to go back to the start of the loop.

```
mystring = "Sammy"
for letter in mystring:
    if letter == 'a':
        continue
    print(letter)
#####
#OUTPUT > S
#m
#m
#y
```

In this example, we've defined `'a'` as the trigger for the loop to restart, as such `'a'` will not be printed.

`break` ends the loop. If we were to replace `continue` with `break` in the example avobe, it would only print `S`, as the letter `'a'` breaks (ends) the loop.

Useful Operators and Functions

Range

The `range()` operator can create ordered ranges of integers. For example:

```
for num in range(10):
    print(num)
#####
#OUTPUT > 0
#1
#2
#3
#4
#5
#6
#7
#8
#9
```

You can also specify a starting integer for the range. For example, if we defined `range(3,10)` into the range function above, it would print all whole integers from 3-9.
Additionally, you can add a step size similar to indexing operations. `range(0,10,2)` would return `0, 2, 4, 6, 8`.

You can also create a **List** with the `.list()` function. Piggybacking onto the range function above, here is an example of casting a list using the `range()` and `list()` functions.

```
list(range(0,11,2))
#####
#OUTPUT > [0,2,4,6,8,10]
```

Enumerate is a function that acts as a counter, specifically with indexing objects within a variable.

```
word = 'abcde'
for item in enumerate(word):
    print(item)
#####
#OUTPUT > (0, 'a')
#(1, 'b')
```

```
#(2, 'c')
#(3, 'd')
#(4, 'e')
```

Note that it outputs tuples, with the index position followed by the item. This means it can be processed with tuple unpacking.

```
word = 'abcde'
for a,b in enumerate(word):
    print(b)
#####
#OUTPUT > a
#b
#c
#d
#e
```

Zip puts together two lists, Let's define two lists:
`mylist1 = [1,2,3]`
`mylist2 = ['a','b','c']`
If we use the zip operator within a for loop, we can begin to combine them.

```
mylist1 = [1,2,3]
mylist2 = ['a','b','c']
for item in zip(mylist1,mylist2):
    print(item)
#####
#OUTPUT > (1, 'a')
#(2, 'b')
#(3, 'c')
```

Note that the Zip function will only zip as much as the shortest list contains. If we were to redefine `mylist1` to have more numbers, it would still only output 3 sets of tuples, as that's the number of objects in the shortest lists - `mylist2`

In
the `in` operator checks if a value is in a list, and will return a boolean value.

```
'x' in [1,2,3]
#####
#Output > False
```

`in` works for all iterable objects.

Min/Max
These functions report the minimum and maximum values of lists.

```
mylist = [1,2,3,4,5]
min(mylist)
#####
#OUTPUT > 1
#####
max(mylist)
#####
#OUTPUT > 5
```

Python has a built-in random library that must first be imported before it can be used.
`from random import shuffle`

Note that after `import` you can press tab to see a list of all importable items from the random library.
With shuffle imported, we can do the following:

```
from random import shuffle
mylist = [1,2,3,4,5,6,7,8,9,10]
shuffle(mylist)
```

Now if we call `mylist` again, it will be in a completely random order.
Similarly, you can import `randint` to generate a random integer from a range.

```
from random import randint
randint(0,100)
```

The output of this code will be any random integer from 0-100.

Input
This operator allows accepting user input.
`input('Enter a number here:')`

This will generate an input box where a user can enter a value. To save this value as a variable for use later in the code, you can cast the input to a variable:

```
result = input('Write a number:')
```

Note that the user input will always output as a string. The user-input data must be cast into its proper type before it can be used.

There are countless other useful methods and functions to be used in Python. You can usually find a few by typing a variable name followed by a period `mylist.` and then pressing tab. You will see a list of methods and functions populated in your IDE that can be used. Additionally, more references can be found at docs.python.org.

List Comprehensions

List Comprehensions are a unique way of creating lists in python. Specifically, it prevents you from having to use a for loop with the `.append()` function. An example of this latter option would be

```
mystring = "hello"
mylist = []

for letter in mystring:
    mylist.append(letter)
#####
#OUTPUT > ['h', 'e', 'l', 'l', 'o']
```

A list comprehension saves space by doing all of this on one line.

```
mystring = "hello"
mylist = [letter for letter in mystring]
#####
#OUTPUT > ['h', 'e', 'l', 'l', 'o']
```

This can be done with any iterable object, including but not limited to strings, variables, ranges, etc.
You can also run additional operations in a comprehension. For example:

```
mylist = [num**2 for num in range(0,11)]
#####
#OUTPUT > [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

You can also add things like if statements after the comprehension.

```
mylist = [x for x in range(0,11) if x%2==0]
#####
#OUTPUT > [0, 2, 4, 6, 8, 10]
```

Intro to Functions

Functions allow the creation of a block of code that can be easily executed many times without needing to constantly rewrite the same code. Creating a function begins with the `def` keyword, which allows you to define a function.

`def name_of_function():`

Of course we begin by defining the function, then using snake casing to name the function. Snake casing is all lowercase with underscores for spaces. Parentheses go at the end. These can be used to pass in arguments and parameters. At the very end is a colon, which indicates indentation. Everything indented after this line of code is inside the function.
Optionally (but conventionally) after the def line is a docstring. This is defined by three single quotes `'''` and is a multi-line comment that describes the function.

```
def name_of_function():
    '''
    Docstring defines the function
    '''
```

Following the docstring is the code that will be run each time you call a function. Note again that everything inside the function will be indented. To call the function, you simply write the function's name with parentheses, but not the colon.

```
def name_call(name):
    '''
    This function prints the passed-in string with a greeting
    '''
    print("Hello "+name)
name_call("Author")
#####
#OUTPUT > Hello Author
```

Note how the last line of code in the above example is *not* indented. This is because we are running the function that we just defined.
This function auto-defines `name`, such that any string you pass into the parentheses of the function call becomes the data for the `name` variable.
Typically, rather than printing directly from the function, we use the `return` keyword, which outputs the result of the function.

Functions follow the syntax of:

```
def say_hello():
    print("hello")
```

Where creating a function begins with the `def` keyword, followed by the name you choose for your function, and then open and closed parentheses. The parentheses are necessary whether you define arguments or not. After the parentheses comes a colon, then a new, indented, line. All code within a function must be indented. This is how you define what code is in a function and what isn't. You can still use nested loops, but they need an extra tab.
When you define your function, you aren't actually running it, just creating it for future use, much like a variable. If you want to actually run your function, you must call the function name and parentheses, but not a colon.

```
say_hello()
###OUTPUT > hello
```

Arguments
Arguments are what go in between the parentheses of your function when you define it. You can name theses anything you want, as they'll be treated as variables that only exist within the function. For example, let's rewrite the previous function:

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

Now we have an argument called `name` and our function needs that argument in order to run.

```
say_hello("Author"):
#####
#OUTPUT > Hello Author!
```

Essentially what we've done here is tell our code that `name = "Author"`, but only within this function. Anywhere we use the `name` variable within this function, it will be replaced with `"Author"`.
If we were to call this function without arguments, Python would throw an error. This is because an argument is required per our definition. We can get around this by providing a default value for the argument.

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

This way if you don't define an argument when you call `say_hello()` later in your code, it will default to `'User'`. If you do define an argument, it overrides the default value. Functions canb e defined with multiple arguments. They must be separated by a comma when defining the function, as well as when calling it.

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

add_num(2,3)
#####
#OUTPUT > 5
```

Let's talk a little more about the `return` keyword. Return essentially is the culmination of a function. In the previous example of `add_num(2,3)`, the code in that function ends with `return`, meaning the returned value (5, in this case) will exist in the place of the function call after it is run.
Another way to look at this is if your function is returning a value, anywhere that function exists will be treated as that value. For example, in the larger scope of our code, the `add_num(2,3)` function will be treated the same as if we had directly input `5` in that position.

NOTE: Ensure that proper data types are being passed into your function args. If you were to call the same function as before but with strings, it would concatenate the strings rather than add numbers.
`add_num('2','3')` will return `'23'` rather than the likely intended `'5'`.

Logic with Functions

The purpose of functions is to have a block of code that can be called quickly at any point. As such all rules of logic and loops apply within a function. As an example, we can create a function to check if a given number is even:

```
def check_even(num):
    if num % 2 == 0:
        return True
    else:
        pass
check_even(2)
#####
#OUTPUT > True
```

This runs a simple if/else statement and returns `True` if a number is even. Since we've created a function, we don't need to write the if/else statement each time, just call the function with the number as an argument and the code within the called function is run automatically.

Another note on `return`,
Whenever a return is called within a function, that is considered the point where the function breaks. If a return condition is met, the function outputs that value and stops running. Notice we

have `pass` in the `else` condition instead of `return False`, this is because if we were to expand the function to accept multiple numbers or lists, the function would break after analyzing just the first input and not all of them.

Let's create a new function to check if any single number in a list is even.

```
def check_even_list(list):
    for num in list:
        if num % 2 == 0:
            return True
        else:
            pass
check_even_list([1,3,6])
#####
#OUTPUT > True
```

In its current state, this function will check if there are any even numbers in the whole list. If there are, it will return `True`. Otherwise it returns nothing.
If we were to add `return False` under the else statement, this function would no longer work properly. It would check the first number, return True or False depending on whether or not the number is even, and then stop running because a return condition had been met.
The workaround for this would be to keep all the current code as-is, then add `return False` *outside* of the for loop at the end.

```
def check_even_list(list):
    for num in list:
        if num % 2 == 0:
            return True
        else:
            pass
    return False
```

This allows the for loop to run first, meaning the loop and function will break if an even number is found. If the for loop completes and its return value is not met, the function defaults to returning False and then breaks. We can take this previous function a bit further by returning a list of all the even numbers, or returning False if there are none.

```
def check_even_list(list):
    evens = []
    for num in list:
        if num % 2 == 0:
            evens.append(num)
        else:
```

```
            pass
    if len(evens) > 0:
        return evens
    else:
        return False

check_even_list([1,2,3,4,5,6])
#####
#OUTPUT > [2, 4, 6]
```

This function begins by defining a temporary variable, `evens`. Variables declared in side of functions are not arguments, and they cannot be called anywhere in the code except within the function in which they were defined. For now, it is an empty list. The function then iterates through the provided list as before, but this time if it finds an even number, it adds it to the evens list using the `.append()` method. Once the for loop is done, it checks the length of the list. If there are more than 0 items, the evens list is returned. Otherwise the function returns false.

Tuple Unpacking With Functions

Let's take a look at working with tuples within functions. I'll create a list made up of tuples, and then write a function to determine which tuple contains the highest number, as well as its associated name.

```
sales_figures = [('Abby',100),('Billy',200),('Cassie',300)]

def employee_check(list):
    current_max = 0
    employee_of_the_month = ''

    for employee,sales in list:
        if sales > current_max:
            current_max = sales
            employee_of_the_month = employee
        else:
            pass
    return (employee_of_the_month,current_max)

employee_check(sales_figures)
#####
#OUTPUT > ('Cassie', 300)
```

Let's analyze what's happening here:

So this function makes use of tuple unpacking, which was discussed in the section **For Loops**. We begin with a list made up of tuples - employees and their respective sales figures. We then define the function to accept one argument (a list made of tuples.)
Inside the function we start with two variables - `current_max` and `employee_of_the_month`. These are going to be u sed to track the values of each tuple, but for now they are placeholder values. `0` and `''` respectively.
We then begin a for loop that unpacks the tuples and checks their values. We defined `employee,sales` as the names of the respective tuple values, so the code checks sales and compares it to the current `current_max` variable. If the checked number is higher, it replaces the `current_max` value. We also have `else: pass`, which does nothing if a tuple is not greater than the current value.
Finally, we return our variables as a tuple. The employee of the month, and their respective sales figures.

We can then unpack this further by just showing the name or sales that are returned by the function. This can be done by creating two variables with a single line of code:
`name,sales = employee_check(sales_figures)`

Now we can either call `name` OR `sales` and find out the individual name or sales figures of the employee of the month.

Functions Application - Three Ball Monte

Let's take a look at the code for a very simple "Three ball monte" game, and go over how it works line by line.

```
from random import shuffle

def shuffle_list(list):
    #This function shuffles and returns the contents of the passed-in list.
    shuffle(list)
    return list

def player_guess():
    #Get the user's guess, cast it as an integer, and return the guess.
    guess = ''

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

def check_guess(list,guess):
    #compare the index position of the shuffled list with the user's guess and
determine win/lose condition
```

```
    if list[guess] == 'O':
        print("You Win!")
    else:
        print("WRONG, FUCKER! GET KILLED IDIOT")
        print(list)

#create the list where the ball lives and run the code
ball = [' ','O',' ']
shuffle_list(ball)
guess = player_guess()
check_guess(ball,guess)
```

First things we should notice are the following:

- We open by importing *shuffle* from python's native *random* library. This was discussed in **Useful Operators and Functions**.
- Three different functions are defined first before any code is actually run.
- After the functions are defined, the code consists of four lines that are run.

Let's tackle this:
`from random import shuffle` - this imports shuffle from Python's *random* library.

First Function - shuffle_list(list):

`def shuffle_list(list):` - we are defining our first of three functions. Note that it takes one argument: a list. Also be sure to read the comment just under this line - it's important to understanding what this line of code does at a glance.

`shuffle(list)` - this shuffles the list that was passed into the function using the random library.

`return list` - this will return the newly shuffled list.

As we can see here, this first function is super simple - it just takes a list as input, shuffles it, and spits it back out.

Second Function - player_guess

`def player_guess():` - once again defining the function. Note that it does not take any arguments. We can also surmise from the title of the function that there will be user input involved. Once again, be sure to read the comment below this line.

`guess = ''` - We begin with an empty string as a placeholder variable. This will ultimately become what the user inputs, but first we need a place to put it. Note that it's a string, as user

input is always processed as a string and must be cast into its appropriate data type. This is discussed in **Useful Operators and Functions**.

`while guess not in ['0','1','2']:` - We are setting up the first part of our while loop. Essentially we are sanitizing user input. If the user inputs anything other than 0, 1, or 2, it's going to keep asking for input. Note also that this is a list of strings, not integers. Once again, the user input will be processed as a string first.

`guess = input("Pick a number: 0, 1, 2 ")` - This is where we gather the user input. The input box will read "Pick a number: 0, 1, 2". We are casting it as a variable for future use.

`return int(guess)` - This returns the user's input *as an integer*. Once the user input is returned from this function, it is converted from a string to an integer for easier processing later on in the code.

Third Function - check_guess

`def check_guess(list,guess):` - Defining our final function here, and this one takes two arguments: the list that was returned from our `shuffle_list` function, and the guess from our `player_guess` function.

`if list[guess] == 'O'` - Perhaps one of the most critical lines of this project. This creates an if/else statement that checks the index position defined by the user's guess (remember it can only be 0, 1, or 2) and if that index position matches the position of 'O' in the shuffled list, the player wins!

`print("You Win!")` This is the win condition of the game. If the user's guess index matches the ball position in the shuffled list, they're told that they win!

`else:` - The lose condition is met when any other input aside from the matching index position is met. Therefore any other guess is incorrect and falls under the "else" condition.

`print("WRONG, FUCKER! GET KILLED IDIOT")` - Prints the lose condition message. Ideally you, the developer, will be standing behind the player with a gun.

`print(list)` - Shows the position of the ball so the user can know just how big of an idiot they were right before they die.

Run The Code

`ball = [' ','O',' ']` - Creates the list to be shuffled by the first function. Its contents are three strings, one of which is a capital O representing the ball. The other two are null spaces, indicating incorrect answers or empty cups.

`shuffle_list(ball)` - Calls our first function, `shuffle_list()`. Remember this function just shuffles the list and spits it back out. Now our ball position has been randomized.

`guess = player_guess()` - Gets the integer of the player's guess by running `player_guess()` and then casts it to a variable for future use.

`check_guess(ball,guess)` - This takes the shuffled list and the collected player input integer and compares them, then returns either the win or lose condition depending on the result.

Conclusion

Overall this is a very easy coding project, and excellent practice for how functions work.

`*Args` and `**Kwargs`

`*args` and `**kwargs` are types of function arguments. They stand for 'arguments' and 'keyword arguments' respectively. Let's say we have a simple function that returns 5% the sum of two inputs:

```
def function_name(a,b)
    return sum((a+b))*0.05
```

This is useful for two inputs, but what if we want to handle more than two? We could just add more inputs in the function definition, but what if the number of arguments is arbitrary?
For this, you would define the function with `*args` in the parentheses.

```
def function_name(*args):
    return sum(args)*0.05
```

all individual parameters passed into a function as `*args` will be automatically cast as a tuple containing these parameters. Note also that the word 'args' is itself arbitrary. You can use any word or identifier here as in regular function definition. As long as it is preceded by an asterisk, it will function as above. That said, convention dictates that you should always use 'args'.

`**kwargs` function in a similar way, only instead of a tuple, it creates a dictionary. Syntax when passing in kwargs looks like this:
`some_function(fruit='apple',veggie='lettuce')`
You are defining the key:value pairings as if they were variables being defined in the function call.

Once again, the name 'kwargs' doesn't matter so much as the double asterisk before it, but convention dictates that 'kwargs' should still be used when this is applied.

Lambda Expressions, Map & Filter Functions

Lambda expressions are "anonymous" functions that are nameless and single-use. To find out why this is useful, we first need to look at the map function.

Map

Let's say we have a function called `square()` that returns the square of an input number. Additionally, we have a list called `my_nums` that contains several integers. We could use a for loop to apply the square function to all the integers in our list individually, or we could use the Map function to apply it globally.
Syntax for the Map function looks like this:
`map(square,my_nums)`
Where the first argument is the function, and the second is the object through which we are iterating.
Running the above code won't produce much, so we have to iterate through it.

```
for item in map(square,my_nums):
    print(item)
```

You could also use `map` as a single-line solution to return a list of all the numbers after they've been through the square function:
`list(map(square,my_nums))`

It's important to note that we aren't actually calling the defined function with `square()` - we are only defining the name of the function inside of map.

Filter

The filter function function returns an iterator yielding those items of the iterable for which when you pass in the item into the function, it's true.
In English, this means that we pass in an object and a function much like before, but the function must return a boolean (True or False). The filter function then iterates through the object, and only returns the items that met the True condition.
For example, let's say we have a function called `check_even` that checks for even numbers. We also have a list called `mynums` that contains all integers between 1-6. We can do the following to cast a list of only `[2,4,6]`:
`list(filter(check_even,mynums))`

Lambda Expressions

Now for why we're here. Let's examine the following function:

```
def square(num):
    return num**2
```

This function can be turned into a lambda expression like so:
`lambda num:num**2`
The differences between the two are noticeable, but they both do the same thing.

- No parenthesis. We define the lambda expression and then immediately give it the parameters it needs.
- No indents or multiple lines. Define the parameter and add a colon, immediately followed by your code to be run.
- No `return` keyword. It's implied that the lambda will always return the result of the run code.
  This is obviously not a replacement for functions., If you have more complex code to run, it's best to turn it into a function.
  We can use lambdas in place of function names for map and filter:
  `list(map(lambda num:num**2,mynums))`

`list(filter(lambda num:num%2==0,mynums))

Nested Statements & Scope

Scope refers to variable assignments in your code, particularly between functions and the general script. For example, let's say we defined a variable at the start of our code - not within a function. In this case, the variable is `x=25`. Now let's say later on in the code we have a variable named `x` that's defined in a function:

```
x = 25

def printer():
    x = 50
    return x
```

Our `x` variable will be handled differently depending on what we call. Running `print(x)` will return 25, while `print(printer())` will return 50. This is because Python has a set of rules to determine scope. This is know as the LEGB rule. Let's break that down:

- Local - Names assigned in any way within a function (def or lambda) and not declared global in that function.
- Enclosing function locals - Names in the local scope of any and all enclosing functions (def or lambda) from inner to outer.

- Global (module) - Names assigned at the top-level of a module ile, or declared global in a def within the file.
- Built-in - Names preassigned in the built-in names module: `open`, `range`, `SyntaxError`, etc.

This is the order in which Python parses variable calls. It starts by looing at at local, then enclosing, then global, then built-in.

- **Local** includes assignments within functions and lambda statements. For example, in `lambda num:num**2` the variable `num` is local only to that expression.
- **Enclosing** is a bit more complex to visualize. Let's start with a block of code:

```
name = 'this is a global string'

def greet():
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()
greet()
#####
#OUTPUT > Hello Sammy
```

As you can see, we have two variables called `name`. One is defined at the start of the code outside of a function, and the other is defined within a function. We are calling the name variable within another function nested inside of `greet`, where our name variable was redefined.
What's happening here is that `name` was not defined in the `hello` function, so Python is searching for any enclosing functions that define `name`. Note that this would produce different results if `greet()` and `hello()` were independent functions rather than being nested.

- **Global** can be best described as a variable that exists outside of a function, whose indentation is all the way to the left. In the previous example, the first `name` variable we defined is a global variable.
- **Built-in** is any function, modifier, name, or otherwise variable that already exists in python. FOr example `len`, `int`, `str`, etc. Be very careful not to overwrite these. If ever you are unsure if a variable name is a built-in python function, you can run `help(name)` If info is returned about that name, it already exists as a built-in property.

Functionally, within the LEGB rule, you can have nested variables share the same name as a global variable without modifying the global. This isn't best practice for readability, but it is possible. However, let's say you wanted to modify that global variable and have it permanently changed from within a function. This is possible with the `global` keyword.

```
x = 50
    def func():
        global x
        x = 'NEW VALUE'
func()
print(x)
#####
#OUTPUT > NEW VALUE
```

Note here that `x` starts with a value of 50. We pull this global variable into `func()` with `global x`, then change its value to a string that says `'NEW VALUE'`. We then run `func()`, and then print ONLY `x`, not `func(x)`. Note that the global value of x has been changed.

Object-Oriented Programming

Intro

Often abbreviated "OOP", Object-Oriented Programming allows for creating your own objects that have methods and attributes. Previously, we've used methods such as `.append()` or `.list()`. These methods use information about the object, as well as the object itself to change or return results about the current object. As such, OOP allows programmers to create their own objects, which allows for the creation of code that's repeatable and organized. As python projects become larger, more OOP will be required to maintain organization and repeatability.

OOP Syntax looks like this:

```
class NameOfClass():

    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2

    def some_method(self):
        #perform some action
        print(self.param1)
```

Lots to note here. Firstly, we begin with the `class` keyword. This is the same thing as an object, and is interchangeable in terminology (but not in actual code)
Next, note the camel case used in the class name. `CapitalizedLettersNoSpaces`. This is conventional best practice, just like `snake_case` in function names. Moving to the next line, note how we have `def __init__(self,param1,param2)`. While this looks like a function definition, it's called a *method* when used inside of a class. This particular `init` method allows us to create an instance of the actual object. *note that the underscores surrounding the word 'init' are not optional.*
We create the object with the `self` keyword, as well as parameters that we will be expected when an instance of the object is created. When we pass these parameters into the `init` method, we assign them to an attribute of the class. Then, we can create other methods within the class, ensuring to pass the `self` keyword to specify that it's a method as part of the greater class.

When first defining `__init__`, we must first pass in the argument of `self`. This is to refer to *this* copy/instance of the object for the duration of the code. Technically any keyword could be used here, but by convention we *must* use `self`. Next we pass in any other arguments to be defined as attributes within the method. Let's take a look at a more easy-to-understand example:

```
class Dog():

    def __init__(self,breed):
        self.breed = breed

my_dog = Dog(breed='Lab')

print(my_dog.breed)
#####
#OUTPUT > Lab
```

So what are we doing here? First, we create the class named `Dog`. Then, in our initialization method, we specify two arguments: `self`, which is required to instantiate the object, and `breed`, which is used later in the method. On the next line, we define that the `.breed` attribute is defined by the `breed` argument via `self.breed = breed`. Next, outside of the class definition, we create a variable of the `Dog` class, and define the breed attribute as the string `'Lab'`. Now, if we ask for the `breed` attribute of our `my_dog` variable, it will give us `'Lab'`.

Of course you can add multiple attributes the same way. Add multiple arguments, and then define them in the `__init__` method with `self.arg = arg` where `arg` is whatever name you give it.

***NOTE** that if your attributes are meant to be specific data types, you must specify that using documentation, comments, or docstrings. There is no way to ensure that any of your attributes must follow a certain data type guideline. For example, in our `Dog` class, it is technically possible to set our `breed` attribute to something other than a string, say `200`. Now every time you call `Dog.breed`, it will return the integer `200`, which could break our code if we are expecting a string.*

Class Object Attributes

These are attributes that are assigned to a class at the highest level so that all instances of this calls have this attribute predefined. These go *after* your `class` keyword call, and *before* your `__init__` method definition. These are defined similar to variables. Let's continue building our `Dog` class with a Class Object Attribute:

```
class Dog():

    species = 'mammal'

    def __init__(self,breed)
        self.breed = breed
```

Class Object Attributes do *not* use the `self` keyword, because they affect *every* instance of the class and not just one.

Methods

Let's talk more about these. Methods are essentially functions that live within a class, and often use the attributes of a class to accomplish their goal. Let's clean up our `Dog` class and continue building:

```
class Dog():

    species = 'mammal'

    def __init__(self,breed,name)
        self.breed = breed
        self.name = name

    def bark(self):
        print("WOOF!")

my_dog = Dog('Lab','Frankie')

my_dog.bark()
```

```
#####
#OUTPUT > WOOF!
```

So the first several lines of code are familiar to us. We've created a class named `Dog`, set a class object attribute defined as `species = 'mammal'`, created our `init` method, and assigned attributes based on arguments. This time we've added a name attribute, and created a method called `bark`. Note that this does use the `self` keyword. This is a super simple method that prints `WOOF!` when called.
After this, we create a variable that is our Dog class type, and define its attributes. Note that we don't need to specify `breed='Lab'`, we can just pass in arguments like a function. Finally, we call our newly created method the same way you could call any other method such as `.upper()` or `.append()`. We use the syntax of a dot, name, and open/closed parentheses. `my_dog.bark()`.

If we wanted to have this dog bark its own name, we would change the print statement to `print("Woof! My name is {}!".format(self.name))` (similarly, we could use an f-string rather than print string formatting. This is up to personal preference.) Note how we use `self.name` for self-referential attributes rather than just `name`. This is because `name` is an attribute recognized by the class, and hasn't been specified in the method arguments.

If we wanted to use an argument without the `self` keyword or a pre-defined attribute, we would write our method like this:

```
    def color(self,col):
        print(f"I am a {col} dog!")
```

Next, we would call `my_dog.color("gray")` which would print `I am a gray dog!` Note how we need to define `col` as its own argument, as well as pass in a string containing the dog's color when calling the method.

A couple quick notes:

- Attributes don't necessarily need to be defined strictly from the arguments defined in the `__init__` method. For example, we can create multiple attributes from just one argument:

```
class Circle():

    def __init__(self,radius):
        self.radius = radius
        self.area = (radius**2) * 3.14
```

in this case, `area` isn't an argument, but it's an attribute we can create from other logic.

- Later in your class code, when you reference a class object attribute, you don't necessarily need to use the `self` keyword. You can instead use the name of the class, such as `Cirlce.pi` (or whatever your class name and class object attribute are called) This helps with code readability, especially in larger projects.

Inheritance and Polymorphism

Inheritance is a way of creating new classes by using classes that have already been defined. This is useful for reusing code that you've already written and reducing the complexity of your program.

Let's create a base starter class:

```
class Animal():

    def __init__(self):
        print("Animal Created")

    def who_am_i(self):
        print("I am an animal")

    def eat(self):
        print("I am eating.")
```

Now that we have this base class, we can revisit our dog class from before, having it inherit the properties of the Animal class

```
class Dog(Animal):

    def__init__(self):
        Animal.__init__(self)
        print("Dog Created")
```

By passing another class into the parentheses of the `Dog` class, we have created a derived class, which inherits attributes from the base class. In our `__init__` method, we also call the Animal `init` method, which creates an instance of the `Animal` class whenever the `Dog` class is created.
This means that whenever we use the `Dog` class in our code, we are also able to call any of the `Animal` methods. If you want to overwrite one of the `Animal` methods within `Dog`, you can do so by simply creating a new method of the same name in the `Dog` class.

Polymorphism refers to the way in which different object classes can share the same method names. Let's say we have two different classes - `Dog` and `Cat` - that do not inherit from each

other, but they both have methods called `.speak()`. Our `Dog.speak()` method prints `woof!` and our `Cat.speak()` method prints `meow!` With Polymorphism, we can write code to call the shared method names easily. One way would be with a for loop:

```
for pet in [mydog,mycat]:
    print(pet.speak())
```

What this demonstrates is that we can build loops/functions around classes and assign them as an argument (in this case, `pet`) for ease of use. A more useful approach would be to use a function:

```
def pet_speak(pet):
    print(pet.speak())
```

This way we can pass any class with the `speak` method into this function and get the results of its individual `.speak()` method.

Special Methods

Special Methods allow us to use built-in operations in python such as `len()` or `print()` with our own user-created objects. For example: normally we are able to use `.len()` on, say, a list, but if we tried it with a class, it would return an error.
Let's create a class as an example:

```
class Book():

    def __init__(self,title,author,pages):
        self.title = title
        self.author = author
        self.pages = pages
```

We can now set a variable as an instance of this class: `b = Book('Python Rocks', 'Jose', 200)`
Now if we were to try and run `print(b)`, all it would return is the memory location of our Book class. We can fix this with a special method.
Special methods are those that are surrounded by double underscores. We've seen one already with `__init__`. Special methods often have a unique role to play in the class and aren't used the same way as regular methods without the underscores.
Going back to our `Book()` example, we can add the `__str__` method. This method allows us to define a string that will represent the class when necessary. Let's add it to our `Book()` class:

```
class Book():

    def __init__(self,title,author,pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} is by {self.author}"
```

Now if we run `print(b)`, we will get `Python Rocks is by Jose`. A similar outcome can be achieved with the `__len__` method. This is what will return when you run `len()` on your class. Let's add it:

```
class Book():

    def __init__(self,title,author,pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} is by {self.author}"

    def __len__(self):
        return self.pages
```

Now in this case, running `len(b)` will return `200`.

You can delete an instance of a class using the `del` keyword. In this case, we could use `del b`. If you wanted an action to take place upon deletion, you would specify the `__del__` method.

```
class Book():

    def __init__(self,title,author,pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} is by {self.author}"

    def __len__(self):
```

```
        return self.pages

    def __del__(self):
        print(f"{self.title} has been deleted.")
```

Now running `del b` would delete our `Python Rocks` book from memory, and print `Python Rocks has been deleted.`
There are many more special methods, and it is advisable to familiarize yourself with them.

Modules & Packages

Pip Install & PyPi

PyPi is a repository for open-source, third-party python packages. Up until this point, we've only used the standard libraries that come internally with python. There are many other libraries available that people have open-sourced and shared on PyPi.

PyPi packages are installed via command line, not your programming IDE. This means that you will need to use command prompt on Windows, or Terminal on Mac/Linux. You can install packages with the command `pip install <package>` where `<package>` is the name of whatever python package you want to install.

If you are unsure exactly what module or package you need in order to accomplish a task, you can always google what python libraries are available for your task. From there you will find documentation on installing and importing that library.

Creating Your Own Modules And Packages

Modules are just `.py` files that you can call in another `.py` script, and packages are a collection of modules. When a package is created, there is an essential `__init__` script that must be created and stored in your package directory (the folder where all your modules are stored) in order for the package to behave as intended.

For example, let's say we have two different `.py` files saved in the same folder:

```
#mymodule.py
def my_func():
    print("Hey, I am being called from mymodule.py!")
```

And another file entirely, saved in the same folder:

```
#myprogram.py
from mymodule import my_func
```

```
my_func()
```

Now we can run `myprogram.py` from the command line.
NOTE: I am creating test environments using Linux, so henceforth I will be referring to 'command prompt' or 'command line' as 'Terminal' and using Linux terminal/Bash to accomplish actions from the Terminal. Adjust as needed for your OS/Command line.

```
cd path/to/file
python3 myprogram.py
```

Note here that we need to have our Terminal open in the folder where our `.py` file is saved, and then call `python3` in order to run it. If done correctly, this should print "Hey, I am being called from mymodule.py!"

When a project becomes sufficiently large, even modules saved in the same folder aren't enough. In this case, we must create a package which contains multiple subfolders full of modules. In all subfolders, we need a file named `__init__.py` This script is completely blank, and only serves to indicate directories in a package.

Let's say we have a directory structure that looks like this:

`📁main_folder`
`mymodule.py`
`myprogram.py`
`📁MyMainPackage`
`__init__.py`
`some_main_script.py`
`📁SubPackage`
`__init__.py`
`mysubscript.py`

With this directory structure, I am able to import and call functions within `myprogram.py` from every subfolder in this list. Syntax looks like this:

```
#myprogram.py
from MyMainPackage import some_main_script
from MyMainPackage.SubPackage import mysubscript
```

Let's note the imports first - `myprogram.py` is at the root of this project folder. To import a script that's one directory deeper in the file structure, we call `from <subfolder> import <script>` Note also how we are importing an entire script from a folder rather than one function from a script. Next, we go another subfolder deeper by using a dot.

`from <folder>.<subfolder> import <script>`
Now that we have these imported, we can call functions from these scripts using a similar dot syntax.
`mysubscript.function_name()`

`__name__` and `'__main__'`

Sometimes when you're working with larger python scripts, it's not uncommon to find the following line of code at the bottom:
`if __name__ == '__main__':`

There is a built-in variable in python called `__name__` that is automatically assigned to a script when it is run. If you run a script from the terminal (Or anywhere else python scripts can be run from) then the `__name__` variable of the currently running script is automatically set to `"__main__"`.

That being said, the above line of code specifies that if the current script is the one called and currently running, then certain actions (specified by the developer) should run. Conventionally, the way this is used is by defining all of your classes, functions, etc. above this line of code, and then assigning your logic under `if __name__ == "__main__":`

Errors and Exception Handling

Intro

Error handling is a way to account for unexpected use cases for your code. Typically when there is an error, the entire script stops. Error handling allows the script to continue running even if there is an error. There are 3 main keywords associated with python error handling:

- `try` - This is the block of code to be tried. (May result in an error)
- `except` - This block of code will execute in the event of an error in the `try` block.
- `finally` - A final block of code to be executed regardless of error.

These keywords function similarly to if/else statements, where `try` is a block of code to be checked first, and if there are errors, it will default to `except`

```
def quick_sum(num1,num2):
    print(num1+num2)

quick_sum(10.'10')
```

Normally the above block of code would fail, because Python can't add an integer and a string. But with error handling, we can continue to run our code.

```
def quick_sum(num1,num2):
    print(num1+num2)

try:
    quick_sum(10+'10')
except:
    print("Hey, it looks like you aren't adding right!")
```

Now instead of crashing due to the `int+string` error, our code will default to printing "Hey, it looks like you aren't adding right!"
You can also add an `else` statement at the bottom which will default when there are no errors.

```
def quick_sum(num1,num2):
    print(num1+num2)

try:
    quick_sum(10+'10')
except:
    print("Hey, it looks like you aren't adding right!")
else:
    print("Math Successful!")
```

Now let's get a little more advanced with specific `except` statements and `finally` statements:

```
try:
    f = open('testfile','r')
    f.write("Write a test line")
except TypeError:
    print("There was a Type Error!")
except:
    print("There was a general error!")
finally:
    print("I always run!")
```

Let's break down what's going on here:
In the `try` block, we are attempting to open a file as type `'r'` and then write to it. This will give a TypeError, as `'r'` is read-only. In our first `except` block, we specify what to do specifically when a TypeError is received. In the second `except` block, we don't specify any error types, which means this block will execute if any other type of error is received. Finally, we have the `finally` block, which will always run regardless of whether or not there was any type of error.

A practical example of error handling would be to nest it inside a while loop.

```
def ask_for_int():
    while True:
        try:
            result = int(input("Please type a whole number"))
        except:
            print("That is not a whole number.")
            continue
        else:
            print("Thank you!")
            break
        finally:
            print("I am here also")
```

Calling `ask_for_int()` will prompt a user for a whole number until one is provided thanks to the while loop. If anything but an integer is provided, it ask again thanks to `continue`. If an integer is provided, the loop ends thanks to `break`. Regardless of the input, it will always print "I am here also" because of the `finally` statement.

Unit Testing

As you begin to expand into larger, multi-file projects, it becomes increasingly important to have tests in place. This way, as you expand and update your code, you can run your test files to make sure previous code still runs as expected.

One method we can use for this is an external library called `pylint`. Since it's external, it must first be installed via the terminal:

```
sudo apt install pylint -y
```

Once you've installed pylint, you can test your code very similarly to the way you'd run it - open a terminal in your script's containing folder and run:

```
pylint <script>.py -r y
```

Running this command will outpout a lot of data about your code. Some of the most important lines are the first few right after the command is called. It will point out how many styling/conventional errors are present, as well as errors in your code. In addition, you can see the number of classes, functions, etc. in your code, how often they're called, duplicated lines, etc. This tool is primarily useful for collaboration.

The Unittest Library

This library allows you to write your own test conditions and run them against your scripts. Test scripts with unittests are written as their own `.py` files saved to the same directory as the script you are trying to test. Best practice is to write your test cases in order from simplest to most complex.

Let's say we have a very simple script that capitalizes the first letter of every word in a string:

```
#cap.py
def cap_text(text):
    return text.title()
```

Now we can write our own test cases for it in another script saved to the same directory:

```
import unittest
import cap

class TestCap(unittest.TestCase):

    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)
        self.assertEqual(result,'Python')

    def test_multiple_words(self):
        text = 'monty python'
        result = cap.cap_text(text)
        self.assertEqual(result,'Monty Python')

if __name__ = '__main__':
    unittest.main()
```

Let's take a line-by-line look at what this script is doing:

1. We are importing Python's native `unittest` library to run our tests.
2. We import the script to be tested.
3. We create a `TestCap` class which inherits from unittest's `TestCase` class.
4. We define our first method to test a single word.
5. Defining the actual text to be used in the test. In this case, 'python'.
6. Setting the `result` variable to our `cap_text` function from the test script, and passing in our previously defined `text` vairable.
7. Asserting that the test will pass if the `result` variable is equal to 'Python'
8. Creating a new method to test multiple words.

9. Setting our `text` variable to a multi-word string - 'monty python'
10. Once again calling our function to be tested and passing in our `text` variable.
11. Asserting that if the result is equal to 'Monty Python', then the test passes.
12. Checking that this is the main running script
13. Running our test using the unittest `main()` function

We can then run this script as we would any other, only instead of printing our variables or calling functions directly, it returns the pass/fail state of each of our defined tests, as well as details of each test.

Python Decorators

Decorators allow you to 'decorate' a function. In this case, to decorate a function means to add additional functionality. If we have a basic function, it can only do what we've programmed it to do. But let's say later on we need to use that function, but add extra steps. As of now, our only options are to either rewrite our original function (which could break our code if it gets used elsewhere) or write an entirely new function that copies all of the code of our previous function, then adds the additional basic functionality.
Decorators exist to allow you to tack on extra functionality to an already existing function. Decorators are denoted with an ampersat `@` on top of the original function.

As a prerequisite, let's set up a function that returns other functions:

```
def hello(name="Author")
    print('The hello() function has ben executed.')

    def greet():
        return '\t This is the greet() function.'

    def welcome():
        return '\t This is the welcome() function.'

    if name == 'Author':
    return greet
    else:
    return welcome
```

The basic premise here is that you can assign the `hello` function to a variable and its output will be one of the two internally-defined functions depending on what you pass in for `name`. For example, `"Author"` is our default name. So if we don't specify another name, we could assign a variable like so
`my_new_func = hello()`
Now `my_new_func()` will return the output of `hello()` when called.

You can also pass a function into another function:

```
def hello():
    return 'Hello!'

def other(some_func):
    print('Other Code Runs Here!')
    print(some_func())

other(hello)
```

The output of this code will be as follows:

```
Other code runs here!
Hello!
```

A critical thing to note in the above code and all code in this section thus far is the careful and deliberate placement of parentheses, particularly after function names, and variables that call functions. When a function is referenced in code without its parentheses, it will not execute. It is simply being referenced. In the case of the previous example, we *referenced* the `hello` function when we passed it into `other`, but we don't execute it (run its code) until we call it in the final print statement of `other`. In this way, we can use functions as arguments rather than their output.

Finally, we can talk about decorators. Let's set up two functions:

```
def new_decorator(original_func):
    def wrap_func():
        print('Some extra code BEFORE')
        original_func()
        print('Some extra code AFTER')
    return wrap_func

def func_needs_decorator():
    print("Decorate Me!")

defcorated_func = new_decorator(func_needs_decorator)

decorated_func()
```

This code outputs the following:

```
Some extra code BEFORE
Decorate Me!
Some extra code AFTER
```

In this example, we have a very long and tedious line of code establishing our `decorated_func` variable. With decorators, we can remove this entirely by simply adding one line before our `func_needs_decorator` definition.

```
@new_decorator
def func_needs_decorator():
    print("Decorate Me!")
```

We still define the `new_decorator()` function above this, but when we define `@new_decorator` *above* a function, any time we run the function below the decorator (in this case, `func_needs_decorator()`) it will automatically run as if it's being passed in as an argument in the decorator. Now we don't need to define or call our `decorated_func` variable. Instead, running `func_needs_decorator()` will produce the exact same output as before. To disable the decorator, just comment out `@new_decorator` or remove that line of code.

Decorators are typically used in python web frameworks such as Django or Flask, and are useful in rendering certain parts of a website as you are developing it.

Python Generators

A generator in Python is a special type of function that is able to return a value, and then later resume where it left off. The main difference in syntax between a function and a generator is the use of a `yield` statement instead of `return`. When a generator function is compiled, they become an object that supports an iteration protocol. This means that when they are called in your code, they don't just return a value and then exit.
Generator functions will automatically suspend and resume their execution and state around the last point of value generation. The advantage is that instead of having to compute an entire series of values up front, the generator computes one value, and waits until the next value is called for.
For example, the `range()` function doesn't produce a list in memory for all the values from start to stop. Instead, it keeps track of the latest number and step size to provide a flow of numbers. Let's explore how these work by starting with a non-generator function:

```
def create_cubes(n):
    result = []
    for x in range(n):
        result.append(x**3)
```

```
    return result

create_cubes(10)
```

This example function returns all cubed numbers from 1 to the user-specified number `(n)`. It does this by calculating the results, storing them in a list, and then returning the list. This can get very memory-intensive, especially as `n` gets larger, because every number is calculated up front and stored in memory. We can turn this function into a generator by removing the list, and using the `yield` keyword instead of `return`.

```
def create_cubes(n):
    for x in range(n):
        yield x**3

for y in create_cubes(10):
    print(y)
```

This turns our function into a generator, allowing us to generate values on the fly rather than storing them in memory. Note that if you were to simply call `create_cubes(10)`, you wouldn't receive the output. You need to iterate through generators using loops much like you would the `range()` function.
There are two functions that are crucial to understanding generators: `next()` and `iter()`.
In our example above, let's say we've assigned the following variable: `g = generate_cubes(10)`.
If we then call `next(g)`, we will get the next generation of the `create_cubes()` function. For example, if we've already generated output from `create_cubes()` 5 times in our code, then calling `next(g)` will output `125`, as it is 5^3 (recall that the first number in a range is 0 by default.) Once the function has maxed out the provided range, attempting to call `next()` again will produce a `StopIteration` error.

The `iter()` function allows you to iterate through a non-generator object. For example, let's say we have a string assigned to a variable: `s = 'hello'`.
Calling `next(s)` would give an error, as a string is not a generator. To fix this, we can assign `s` as an iterable object with `iter()`. `t = iter(s)`.
Now we can call `next(t)` to iterate through the letters in our `hello()` string as if it were a generator.

Advanced Python Modules

The following are several advanced modules built into python that have gone mostly unmentioned thus far in these notes. The next section will be dedicated to these modules and

some of their functionality. Note that I will not be covering *all* of the functionality of these modules, so independent research will be crucial in uncovering all that they do.

Python Collections Module

The Python Collections Module implements specialized container data types. A container in base python would be something like a dictionary or a tuple - a data type that stores other data. The collections module has more specialized containers. As is the case with specialization, the Container module won't always be necessary, but it can come in handy for certain use cases. One such useful object is the Counter class. Let's say we have a list:
`mylist = [1,1,1,1,2,2,2,2,2,2,3,3,3,3,3]`
If we wanted to count the number of each unique item in this list, it would require a fairly complicated function that creates a dictionary, iterates through the list, and for each object, either create a new key or iterate the value +1 for each object. No fun.
Instead, we let the Counter do all the work for us.

```
from collections import Counter
Counter(mylist)
```

Assuming our pre-defined `mylist` is what we pass into `counter`, the output would be:
`Couter({2:6,3:5,1:4})`
So as you can see, Counter automatically returns a dictionary of each unique element, as well as the number of times it occurs in the passed-in object.
Counter can also count letters in a string, and even words in a sentence. To do the latter, you'll need to make use of the `.split()` function. (Recall that `.split()` will separate a string into unique strings based on the value passed into its parentheses. If none is specified, it uses spaces.)

```
sentence = "How many unique words are in this unique sentence?"
```

Now we can call `Counter(sentence.split())` and get a count of each unique word.

Another handy object from the Collections library is `defaultdict` - which is useful for assigning new keys in a dictionary quickly. Let's look at a python standard dictionary first:
`d = {'a':100}`
Right now, calling `d[a]` would return a valid output of `100`. But if you were input any other value to call d - say `d['INCORRECT']`, you would receive an error. Specifically, a `KeyError`.
With `defaultdict`, you have the option to set a default value, so that if you pass in a key that doesn't exist, it simply create that key with the new default value rather than giving an error.
Syntax looks like this:

```
from collections import defaultdict
d = defaultdict(lambda:0)
```

```
d['WRONG']
```

The output of this code would be `0`.
It is important to note that the `lambda` keyword is necessary in defining the default value. You don't have to write the actual lambda expression - this is pre-defined in the module class. Now, any time you pass in a key expression that doesn't currently exist in the dictionary, a new key will be created with our pre-defined value. In this case, `0`.

`namedtuple` sets out to improve tuple readability by adding name values to each index position of a given tuple object. Creating a named tuple looks similar to building a class:

```
from collections import namedtuple
Dog = namedtuple('Dog',['age','breed','name'])
whiskey = Dog(age=5,breed='Dachsund',name='Whiskey')
```

Note how we first create a type similar to a class- in this case, `Dog`. Once we specify the tyhpe and all attributes or other data we'd like to name and store, we create the tuple by calling the type (Dog) and specifying all info. Now if we ant to call any of these values, we simply call the following:
`whiskey.age`
In this case, it will output `5`.
You still have the option to call index values such as `whiskey[0]`, but if your tuple is particularly large, it can be useful to store its data in this way.

I will say this for all of these libraries, but I highly recommend looking at the documentation for each of them on your own. There are some truly useful things to be found within python's Collections module.

Python OS & Shell Utilities Modules

These modules focus on functionality for opening, reading, and writing files and folders on your computer or the computer on which you run your script. We have established previously that it is possible to create new text files with `open()` and `write()`. For example:

```
f = open('practice.txt','w+')
f.write('This is a test string')
f.close
```

This creates a simple `.txt` file in whatever directory you open and run this script. The OS module allows us to modify and navigate the file system of the computer on which you are running the script. You can get your current filepath with the following:

```
import os
os.getcwd()
```

If you run `os.listdir()`, it will create a list object of every file and folder in the current directory. You can also specify which directory you'd like to ist by passing the filepath as a a string into the `listdir()` function. Recall that file structure is represented by a double backslash in windows, such as `C:\\Users\\user1`. Linux and MacOS use single slashes such as `~/home/Documents`

You can use the Shell Utilities module (`shutil`) to move files and folders around. Let's assume our script is in the same directory as our `practice.txt` file from before:

```
import shutil
shutil.move('practice.txt','C:\\Users\\user1')
```

This will move the practice file into the `user1` folder. Note how we specify a target (file to be moved) and a destination (location to move it to) both represented as strings separated by a comma.

There are 3 main ways to delete files using `os` and `shutil`:
`os.unlink('filepath')` - this deletes a single file at the specified path.
`os.rmdir('path')` - this deletes an *empty* folder at the provided path.
`shutil.rmtree('path')` - This deletes *all* files and folders at the specified path. **USE CAREFULLY**

IMPORTANT! if you use any of these methods of deleting files/folders, they will become unrecoverable. Files/folders deleted in this manner are deleted permanently rather than being sent to trash. Fortunately, there is a module for this. At your command line, install the `Send2Trash` module. As always, I am using Linux:

```
pip install send2trash
```

Now as long as you `import send2trash` in your script, you can remove a file by sending it to trash rather than outright deleting it.

```
import send2trash
send2trash.send2trash('practice.txt')
```

You can specify any file here as long as it is in the current directory.

A very useful method from the `os` module is `walk()`. It requires on argument, which is the 'top-level' folder you want to examine. `walk()` looks at the directory you've provided, and indexes all files, folders, and subfolders branching from it until it has no more subfolders to open.

```
import os

for folder, subfolder, file in os.walk(os.getcwd()):
    print(f'currently in {folder}')
    print('\n')
    print('The subfolders are: ')
    for sub_fold in subfolder:
        print(f'\t Subfolder: {sub_fold}')
    print('\n')
    print('The files are: ')
    for f in file:
        print(f'\t File: {f})
    print('\n')
```

This script will print the names and contents of every file, subfolder, sub-subfolder, etc. until it has gone through everything in the specified filepath.

Python Datetime Module

The `datetime` module allows for operations regarding date and time, time zones, elapsed time, and operations between date/time models.
We'll start with the simplest function: `time()`. This module works on a 24-hour clock system, so inputting `2` will default to 2:00 AM. The time function accepts inputs for hour, minute, second, and microsecond (in that order) as well as timezone info. Let's take a look:

```
import datetime
mytime = datetime.time(2,20)
```

Since we only gave it two arguments, `mytime` defaulted them to 2 hours and 20 minutes. We can call `mytime.hour` which will output `2`, and `mytime.minute` will output `20`. Calling `print(mytime)` will print `02:20:00`. Anything undefined will automatically be assigned a zero, and microseconds will only print if they have a value greater than 0.
This `time()` object only contains time data, and has no info on the current (or otherwise specified) date. To store this info, we would use a `date()` object, or a combined `datetime()` object. Let's start with `date()`.
The arguments accepted by `date()` are, in order, Year/Month/Day. So if you wanted to input the date of writing this page of notes, it would be `datetime.date(2025,8,18)`. Alternatively,

you have the option of pulling the current date with `datetime.date.today()`.
If you wanted to store the date *and* time in a single object, you would use a `datetime` object. Since that's also the name of the entire module, you import this one a little differently:

```
from datetime import datetime
mydatetime = datetime(2021,10,3,14,20,1)
```

Now printing `mydatetime` returns `2021-10-03- 14:20:01`. The best way to remember the order of arguments for `datetime` is to go from largest to smallest:
Year/Month/Day/Hour/Minute/Microsecond/Timezone Info.
You can also replace currently defined values with `mydatetime.replace(year=2025)`.
You can do arithmetic between dates/times by using either a `date()` or `datetime()` object.

```
from datetime import date

date1 = (2021,11,3)
date2 = (2020,11,3)
```

Right now, if we were to call `date1-date2`, we'd get an object unique to the `datetime` module: `datetime.timedelta(365)`. The `timedelta` object has its own useful methods. For example, if we turned the above into a variable: `result = date1 - date2` we can call `result.days` to output the number of days in this `timedelta` object. In this case, 365.

We can perform similar arithmetic with `datetime` objects.

```
from datetime import datetime

datetime1 = datetime(2021,11,3,22,0)
datetime2 = datetime(2020,11,3,12,0)
```

These two dates are 1 year and 10 hours apart. The resulting `timedelta` from subtracting `datetime1 - datetime2` would present this in days and seconds. `timedelta(365,36000)`.
You could then do some simple math do find out how many hours this is. Continuing the code from above:

```
mydiff = datetime1 - datetime2
hours = mydiff.seconds/60/60
```

As a final note, you can also call `.total_seconds()` on a `timedelta` object to output the entire time as seconds.

Python Math & Random Modules

These modules contain useful methods for doing math and generating randomness. We've already used `random.shuffle` in previous projects if you've been following along, but here we will expand on some of the other uses for these modules.
The math module is worth researching for the vast amount of useful mathematical methods it has. We will go over a few of them here:
Let's say we have a value: `value = 4.35` We can use `math.floor(value)` and `math.ceil(value)` to return the nearest whole number. `floor()` rounds down, outputting `4`, and `.ceil()` rounds up, outputting `5`. Note that `floor()` will *always* round down, and likewise `ceil()` will always round up, regardless of its relative median value. If you want a true rounding operation, you can use `round()` which is built into python and not part of the math module.
The math module also has various well-known constants, such as `math.pi`, `math.e`, `math.inf`, and `math.nan` (not a number).
With the `random` library, you can generate a random number from a range with `random.randint(a,b)`, where a & b are the low and high numbers of your range.
You can set a seed at the start of your code so the random library will always produce the same set of random numbers. This is useful for code testing and debugging. You can set this with `random.seed(value)` where `value` can be any integer at all.
You can also pull a random item from a set list. Let's say we have `mylist`, which contains every integer between 0-19. We can run `random.choice(mylist)` and it will return one value at random from `mylist`. This doesn't remove it form the list, it only returns the value.
You can also return multiple values from a list. There are two ways to do so: sampling with replacement, and sampling without replacement. Sampling with replacement means that it's possible to return the same value multiple times. Sampling without replacement means that each random number or value must be unique.
To sample with replacement, we call `random.choices(population=mylist,k=10)` where `population` is where we are pulling from, and `k` is the number of values to be returned. This will return a new list containing all of the randomly selected values.
To sample without replacement, we run `random.sample(population=mylist,k=10)`
Note that the keywords `population` and `k` are not optional in either of these method calls.

Python Debugger (pdb)

Sometimes when you are working with a lot of data, it can be easy to lose track of variables and their values. If you run into an error in your code, you can insert a debug trace on a line just before said error so you can check the values of each of your variables at that exact point in your code. For example:

```
import pdb

x=5
x+=9
pdb.set_trace()
x+=7
```

When you run this code, it will stop where you called `pdb.set_trace()` and ask for an input. When you input a variable name, it will output its exact value as of this point in the code so you can ensure all values are what they should be. You can also use this input box to perform operations on variables, such as `x+y`, etc.

Regular Expressions (regex)

In base python, we have the ability to locate substrings within a string (ex: `if 'dog' in 'dogs are great'...`) but we need to know the exact contents, case, spacing, etc. of the desired substring. We don't have an easy way in base python to search for a patter within a string, such as a phone number or email address.
The `re` library allows us to create a specialized pattern string and then search within text for this pattern. As an example, let's say we are looking for a phone number: `(555)-555-5555`. In regex syntax, that would look like this:
`r"(\d\d\d)-\d\d\d-\d\d\d\d`. The lowercase 'r' outside the quotes indicates to python that this isn't a normal string, and has regecs identifiers in it. The `\d` indicators are wildcards to indicate any given digit. The parentheses and dashes aren't regex identifiers, they are just part of the string. Regex also has quantifiers that can simplify this even further: `r'(\d{3})-\d{3}-\d{4}'` is another way of writing the above example. The quantifiers in curly braces tell regex how many of each indicator (`\d` in our case) to look for.

I highly encourage the independent research of the `re` and `regex` libraries. It is incredibly useful, and holds multitudes of functions - so much so that an entire guide could be published just on these libraries alone. I haven't even begun to scratch the surface here. If there is any library from this guide that you search up, ensure it is this one.

Timing code for efficiency

Python has infinite ways of solving any given problem. If you find yourself in a position where you have multiple different scripts to do the same thing, you can time each one to find out which is the most efficient. This can be done by using the `timeit` module.
`timeit` is a bit odd in its input structure, as you have to input your code to be tested as a string argument inside the `timeit` function. Think of it like this:

```
setup = """
def defining_code() :
    return 'This setup variable is a string defining the functions to be
dested, while keeping proper indentation'
    """

stmt = """defining_code()"""
```

Once you've defined your `setup` and `stmt` (statement) variables, you can time your code with the following:

```
import timeit

#Define your setup and stmt variables here

timeit.timeit(stmt,setup,number=500)
```

The number in the final argument place is the number of times you want to run your code in the test. The resulting output will be the time it took to complete the test in seconds.

Working with .zip files

.zip files are compressed directories that are made to take up less space than their uncompressed counterparts. They are also useful for transferring large amounts of data over the internet without loss of quality.
Let's assume we have two example text files named `fileone.txt` and `filetwo.txt`. These files exist in the directory where we plan to run this script. One way we can compress these .txt files is by using the `zipfile` library.

```
import zipfile
comp_file = zipfile.ZipFile('comp_file.zip','w')
```

Let's pause there. A few things to note:
First, note how the actual function call that we are making is capitalized differently from the library name. This is important and not optional.
Second, the two arguments taken by the `ZipFile()` method are the name of the zip folder (including its extension) and the 'mode.' Mode is similar to when we use the `open()` function, in that `'w'` is write mode.
Finally, when we run just the two lines of code above, it will create our `comp_file.zip` folder in the directory in which the script was run. As of now, this folder is blank. Let's continue where we left off:

```
comp_file.write('fileone.txt',compress_type=zipfile.ZIP_DEFLATED)
comp_file.write('filetwo.txt',compress_type=zipfile.ZIP_DEFLATED)
```

The `compress_type` argument specifies the type of compression to be used. Zip Deflated is the most common one, but you can read the `zipfile` docs to see what all the compression types do. Once we have put all of our files into the .zip folder, we can call `comp_file.close()` to close the folder and make it accessible on our computer.
Extracting a .zip file is a similar process:

```
import zipfile
zip_obj=zipfile.ZipFile('comp_file.zip')
```

Once we've defined our `zip_obj` variable, we have two options:
We can extract a single file from the folder with `zip_obj.extract('filename.txt')`
Or we can create a new, non-zipped folder with all of the contents of the .zip:
`zip_obj.extractall('extracted_content')`
Note that the argument taken by `.extractall()` is the folder name, meaning you can put any string here to name your folder.
These `zipfile` functions also contain argument options to specify a filepath instead of defaulting to the script directory, so as always, be sure to look up this library on your own time.

You can also zip an entire pre-existing folder with `shutil`:

```
import shutil

dir_to_zip = 'path\\to\\file'
output_filename = 'zipfolder_name'
shutil.make_archive(output_filename,'zip',dir_to_zip)
```

Where `dir_to_zip` is the exact filepath to your folder (including the folder itself) and `output_filename` is what the new .zip file will be called. `'zip'` is the compression protocol (as opposed to, say, tar.gz) You can extract with `shutil` using a similar method:

```
import shutil

shutil.unpack_archive('archive_name.zip','unpacked_filenmae','zip')
```

Where the final argument is the archive's filetype.