Numbers and more in Python!

In this lecture, we will learn about numbers in Python and how to use them.

We'll learn about the following topics:

```
1.) Types of Numbers in Python
2.) Basic Arithmetic
3.) Differences between classic division and floor division
4.) Object Assignment in Python
```

Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

Examples	Number "Type"
1,2,-5,1000	Integers
1.2,-0.5,2e2,3E2	Floating-point numbers

Now let's start with some basic arithmetic.

Basic Arithmetic

In [1]:
```
# Addition
2+1
```

Out[1]: 3

In [2]:
```
# Subtraction
2-1
```

Out[2]: 1

In [3]:
```
# Multiplication
2*2
```

Out[3]: 4

In [4]:
```
# Division
3/2
```

Out[4]: 1.5

In [5]:
```
# Floor Division
7//4
```

Out[5]: 1

Whoa! What just happened? Last time I checked, 7 divided by 4 equals 1.75 not 1!

The reason we get this result is because we are using "*floor*" division. The // operator (two forward slashes) truncates the decimal without rounding, and returns an integer result.

So what if we just want the remainder after division?

In [6]:
```
# Modulo
7%4
```

Out[6]: 3

4 goes into 7 once, with a remainder of 3. The % operator returns the remainder after division.

Arithmetic continued

In [7]:
```
# Powers
2**3
```

Out[7]: 8

In [8]:
```
# Can also do roots this way
4**0.5
```

Out[8]: 2.0

In [9]:
```
# Order of Operations followed in Python
2 + 10 * 10 + 3
```

Out[9]: 105

In [10]:
```
# Can use parentheses to specify orders
(2+10) * (10+3)
```

Out[10]: 156

Variable Assignments

Now that we've seen how to use numbers in Python as a calculator let's see how we can assign names and create variables.

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

In [11]:
```
# Let's create an object called "a" and assign it the number 5
a = 5
```

Now if I call *a* in my Python script, Python will treat it as the number 5.

In [12]:
```
# Adding the objects
a+a
```

Out[12]: 10

What happens on reassignment? Will Python let us write it over?

In [13]:
```
# Reassignment
a = 10
```

In [14]:
```
# Check
a
```

Out[14]: 10

Yes! Python allows you to write over assigned variable names. We can also use the variables themselves when doing the reassignment. Here is an example of what I mean:

In [15]:
```
# Check
a
```

Out[15]: 10

In [16]:
```
# Use A to redefine A
a = a + a
```

In [17]:
```
# Check
a
```

Out[17]: 20

The names you use when creating these labels need to follow a few rules:

1. Names can not start with a number.
2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|\()!@#$%^&*~-+
4. It's considered best practice (PEP8) that names are lowercase.
5. Avoid using the characters 'l' (lowercase letter el), 'O' (uppercase letter oh),
   or 'I' (uppercase letter eye) as single character variable names.
6. Avoid using words that have special meaning in Python like "list" and "str"

Using variable names can be a very useful way to keep track of different variables in Python. For example:

In [18]:
```
# Use object names to keep better track of what's going on in your code!
my_income = 100

tax_rate = 0.1

my_taxes = my_income*tax_rate
```

In [19]:
```
# Show my taxes!
my_taxes
```

Out[19]:
```
10.0
```

So what have we learned? We learned some of the basics of numbers in Python. We also learned how to do arithmetic and use Python as a basic calculator. We then wrapped it up with learning about Variable Assignment in Python.

Up next we'll learn about Strings!

Variable Assignment

Rules for variable names

- names can not start with a number
- names can not contain spaces, use _ intead
- names can not contain any of these symbols:

```
:'",<>/?|\!@#%^&*~-+
```

- it's considered best practice (PEP8 (https://www.python.org/dev/peps/pep-0008/#function-and-variable-names)) that names are lowercase with underscores
- avoid using Python built-in keywords like `list` and `str`
- avoid using the single characters `l` (lowercase letter el), `O` (uppercase letter oh) and `I` (uppercase letter eye) as they can be confused with `1` and `0`

Dynamic Typing

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

In [1]:
```
my_dogs = 2
```

In [2]:
```
my_dogs
```

Out[2]:
```
2
```

In [3]:
```
my_dogs = ['Sammy', 'Frankie']
```

In [4]:
```
my_dogs
```

Out[4]:
```
['Sammy', 'Frankie']
```

Pros and Cons of Dynamic Typing

Pros of Dynamic Typing

- very easy to work with
- faster development time

Cons of Dynamic Typing

- may result in unexpected bugs!
- you need to be aware of `type()`

Assigning Variables

Variable assignment follows `name = object`, where a single equals sign `=` is an *assignment operator*

In [5]:
```
a = 5
```

In [6]:
```
a
```

Out[6]:
```
5
```

Here we assigned the integer object `5` to the variable name `a`.
Let's assign `a` to something else:

In [7]:
```
a = 10
```

In [8]:
```
a
```

Out[8]:
```
10
```

You can now use `a` in place of the number `10`:

In [9]:
```
a + a
```

Out[9]:
```
20
```

Reassigning Variables

Python lets you reassign variables with a reference to the same object.

In [10]:
```
a = a + 10
```

In [11]:
```
a
```

Out[11]:
```
20
```

There's actually a shortcut for this. Python lets you add, subtract, multiply and divide numbers with reassignment using `+=`, `-=`, `*=`, and `/=`.

In [12]:
```
a += 10
```

In [13]:
```
a
```

Out[13]:
```
30
```

In [14]:
```
a *= 2
```

In [15]:
```
a
```

Out[15]:
```
60
```

Determining variable type with `type()`

You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include:

- **int** (for integer)
- **float**
- **str** (for string)
- **list**
- **tuple**
- **dict** (for dictionary)
- **set**
- **bool** (for Boolean True/False)

In [16]:
```
type(a)
```

Out[16]:
```
int
```

In [17]:
```
a = (1,2)
```

In [18]:
```
type(a)
```

Out[18]:
```
tuple
```

Simple Exercise

This shows how variables make calculations more readable and easier to follow.

In [19]:
```
my_income = 100
tax_rate = 0.1
my_taxes = my_income * tax_rate
```

In [20]:
```
my_taxes
```

Out[20]:
```
10.0
```

Great! You should now understand the basics of variable assignment and reassignment in Python. Up next, we'll learn about strings!

Strings

Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
3.) String Indexing and Slicing
4.) String Properties
5.) String Methods
6.) Print Formatting
```

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

In [1]:
```
# Single word
'hello'
```

Out[1]:
```
'hello'
```

In [2]:
```
# Entire phrase
'This is also a string'
```

Out[2]:
```
'This is also a string'
```

In [3]:
```
# We can also use double quote
"String built with double quotes"
```

Out[3]:
```
'String built with double quotes'
```

In [4]:
```
# Be careful with quotes!
' I'm using single quotes, but this will create an error'
```

```
  File "<ipython-input-4-da9a34b3dc31>", line 2
    ' I'm using single quotes, but this will create an error'
       ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in `I'm` stopped the string. You can use combinations of double and single quotes to get the complete statement.

In [5]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Out[5]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Now let's learn about printing strings!

Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

In [6]:
```
# We can simply declare a string
'Hello World'
```

Out[6]:
```
'Hello World'
```

In [7]:
```
# Note that we can't output multiple strings this way
'Hello World 1'
'Hello World 2'
```

Out[7]:
```
'Hello World 2'
```

We can use a print statement to print a string.

In [8]:
```
print('Hello World 1')
print('Hello World 2')
print('Use \n to print a new line')
print('\n')
print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

String Basics

We can also use a function called len() to check the length of a string!

In [9]:
```
len('Hello World')
```

Out[9]:
```
11
```

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets `[]` after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called `s` and then walk through a few examples of indexing.

In [10]:
```
# Assign s as a string
s = 'Hello World'
```

In [11]:
```
#Check
s
```

Out[11]:
```
'Hello World'
```

In [12]:
```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [13]:
```
# Show first element (in this case a letter)
s[0]
```

Out[13]:
```
'H'
```

In [14]:
```
s[1]
```

Out[14]:
```
'e'
```

In [15]:
```
s[2]
```

Out[15]:
```
'l'
```

We can use a `:` to perform *slicing* which grabs everything up to a designated point. For example:

In [16]:
```
# Grab everything past the first term all the way to the length of s which is le
s[1:]
```

Out[16]: 'ello World'

In [17]:
```
# Note that there is no change to the original s
s
```

Out[17]: 'Hello World'

In [18]:
```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[18]: 'Hel'

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

In [19]:
```
#Everything
s[:]
```

Out[19]: 'Hello World'

We can also use negative indexing to go backwards.

In [20]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[20]: 'd'

In [21]:
```
# Grab everything but the last letter
s[:-1]
```

Out[21]: 'Hello Worl'

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

In [22]:
```
# Grab everything, but go in steps size of 1
s[::1]
```

Out[22]: 'Hello World'

In [23]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[23]: 'HloWrd'

In [24]:
```
# We can use this to print a string backwards
s[::-1]
```

Out[24]:
```
'dlroW olleH'
```

String Properties

It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed or replaced. For example:

In [25]:
```
s
```

Out[25]:
```
'Hello World'
```

In [26]:
```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-976942677f11> in <module>()
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment
```

Notice how the error tells us directly what we can't do, change the item assignment!

Something we *can* do is concatenate strings!

In [27]:
```
s
```

Out[27]:
```
'Hello World'
```

In [28]:
```
# Concatenate strings!
s + ' concatenate me!'
```

Out[28]:
```
'Hello World concatenate me!'
```

In [29]:
```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

In [30]:
```
print(s)
```

```
Hello World concatenate me!
```

In [31]:
```
s
```

Out[31]:
```
'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

In [32]:
```
letter = 'z'
```

In [33]:
```
letter*10
```

Out[33]:
```
'zzzzzzzzzz'
```

Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

In [34]:
```
s
```

Out[34]:
```
'Hello World concatenate me!'
```

In [35]:
```
# Upper Case a string
s.upper()
```

Out[35]:
```
'HELLO WORLD CONCATENATE ME!'
```

In [36]:
```
# Lower case
s.lower()
```

Out[36]:
```
'hello world concatenate me!'
```

In [37]:
```
# Split a string by blank space (this is the default)
s.split()
```

Out[37]:
```
['Hello', 'World', 'concatenate', 'me!']
```

In [38]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

Out[38]:
```
['Hello ', 'orld concatenate me!']
```

There are many more methods than the ones covered here. Visit the Advanced String section to find out more!

Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

In [39]:
```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

Out[39]:
```
'Insert another string with curly brackets: The inserted string'
```

We will revisit this string formatting topic in later sections when we are building our projects!

Next up: Lists!

Lists

Earlier when discussing strings we introduced the concept of a *sequence* in Python. Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

1.) Creating lists
2.) Indexing and Slicing Lists
3.) Basic List Methods
4.) Nesting Lists
5.) Introduction to List Comprehensions

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

In [1]:
```
# Assign a list to an variable named my_list
my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

In [2]:
```
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

In [3]:
```
len(my_list)
```

Out[3]:
```
4
```

Indexing and Slicing

Indexing and slicing work just like in strings. Let's make a new list to remind ourselves of how this works:

In [4]:
```
my_list = ['one','two','three',4,5]
```

In [5]:
```
# Grab element at index 0
my_list[0]
```

Out[5]:
```
'one'
```

In [6]:
```
# Grab index 1 and everything past it
my_list[1:]
```

Out[6]:
```
['two', 'three', 4, 5]
```

In [7]:
```
# Grab everything UP TO index 3
my_list[:3]
```

Out[7]:
```
['one', 'two', 'three']
```

We can also use + to concatenate lists, just like we did for strings.

In [8]:
```
my_list + ['new item']
```

Out[8]:
```
['one', 'two', 'three', 4, 5, 'new item']
```

Note: This doesn't actually change the original list!

In [9]:
```
my_list
```

Out[9]:
```
['one', 'two', 'three', 4, 5]
```

You would have to reassign the list to make the change permanent.

In [10]:
```
# Reassign
my_list = my_list + ['add new item permanently']
```

In [11]:
```
my_list
```

Out[11]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

We can also use the * for a duplication method similar to strings:

In [12]:
```
# Make the list double
my_list * 2
```

Out[12]:
```
['one',
 'two',
 'three',
 4,
 5,
 'add new item permanently',
 'one',
 'two',
 'three',
 4,
 5,
 'add new item permanently']
```

In [13]:
```
# Again doubling not permanent
my_list
```

Out[13]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

Basic List Methods

If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

Let's go ahead and explore some more special methods for lists:

In [14]:
```
# Create a new list
list1 = [1,2,3]
```

Use the **append** method to permanently add an item to the end of a list:

In [15]:
```
# Append
list1.append('append me!')
```

In [16]:
```
# Show
list1
```

Out[16]:
```
[1, 2, 3, 'append me!']
```

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

In [17]:
```
# Pop off the 0 indexed item
list1.pop(0)
```

Out[17]:
```
1
```

In [18]:
```
# Show
list1
```

Out[18]:
```
[2, 3, 'append me!']
```

In [19]:
```
# Assign the popped element, remember default popped index is -1
popped_item = list1.pop()
```

In [20]:
```
popped_item
```

Out[20]:
```
'append me!'
```

In [21]:
```
# Show remaining list
list1
```

Out[21]:
```
[2, 3]
```

It should also be noted that lists indexing will return an error if there is no element at that index. For example:

In [22]:
```
list1[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-22-af6d2015fa1f> in <module>()
----> 1 list1[100]

IndexError: list index out of range
```

We can use the **sort** method and the **reverse** methods to also effect your lists:

In [23]:
```
new_list = ['a','e','x','b','c']
```

In [24]:
```
#Show
new_list
```

Out[24]:
```
['a', 'e', 'x', 'b', 'c']
```

In [25]:
```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

In [26]:
```
new_list
```

Out[26]:
```
['c', 'b', 'x', 'e', 'a']
```

In [27]:
```
# Use sort to sort the list (in this case alphabetical order, but for numbers it
new_list.sort()
```

In [28]:
```
new_list
```

Out[28]:
```
['a', 'b', 'c', 'e', 'x']
```

Nesting Lists

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

Let's see how this works!

In [29]:
```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]
```

In [30]:
```
# Show
matrix
```

Out[30]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

In [31]:
```
# Grab first item in matrix object
matrix[0]
```

Out[31]:
```
[1, 2, 3]
```

In [32]:
```
# Grab first item of the first item in the matrix object
matrix[0][0]
```

Out[32]:
```
1
```

List Comprehensions

Python has an advanced feature called list comprehensions. They allow for quick construction of lists. To fully understand list comprehensions we need to understand for loops. So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.

But in case you want to know now, here are a few examples!

In [33]:
```
# Build a list comprehension by deconstructing a for loop within a []
first_col = [row[0] for row in matrix]
```

In [34]:
```
first_col
```

Out[34]:
```
[1, 4, 7]
```

We used a list comprehension here to grab the first element of every row in the matrix object. We will cover this in much more detail later on!

For more advanced methods and features of lists in Python, check out the Advanced Lists section later on in this course!

Dictionaries

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

1.) Constructing a Dictionary
2.) Accessing objects from a dictionary
3.) Nesting Dictionaries
4.) Basic Dictionary Methods

So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

In [1]:
```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

In [2]:
```
# Call values by their key
my_dict['key2']
```

Out[2]:
```
'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

In [3]:
```
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [4]:
```
# Let's call items from the dictionary
my_dict['key3']
```

Out[4]:
```
['item0', 'item1', 'item2']
```

In [5]:
```
# Can call an index on that value
my_dict['key3'][0]
```

Out[5]:
```
'item0'
```

In [6]:
```
# Can then even call methods on that value
my_dict['key3'][0].upper()
```

Out[6]:
```
'ITEM0'
```

We can affect the values of a key as well. For instance:

In [7]:
```
my_dict['key1']
```

Out[7]:
```
123
```

In [8]:
```
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

In [9]:
```
#Check
my_dict['key1']
```

Out[9]:
```
0
```

A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

In [10]:
```
# Set the object equal to itself minus 123
my_dict['key1'] -= 123
my_dict['key1']
```

Out[10]:
```
-123
```

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

In [11]:
```
# Create a new dictionary
d = {}
```

In [12]:
```
# Create a new key through assignment
d['animal'] = 'Dog'
```

In [13]:
```
# Can do this with any object
d['answer'] = 42
```

In [14]:
```
#Show
d
```

Out[14]:
```
{'animal': 'Dog', 'answer': 42}
```

Nesting with Dictionaries

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

In [15]:
```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

Wow! That's a quite the inception of dictionaries! Let's see how we can grab that value:

In [16]:
```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

Out[16]:
```
'value'
```

A few Dictionary Methods

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few of them:

In [17]:
```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}
```

In [18]:
```
# Method to return a list of all keys
d.keys()
```

Out[18]:
```
dict_keys(['key1', 'key2', 'key3'])
```

In [19]:
```
# Method to grab all values
d.values()
```

Out[19]:
```
dict_values([1, 2, 3])
```

In [20]:
```
# Method to return tuples of all items  (we'll learn about tuples soon)
d.items()
```

Out[20]:
```
dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

Hopefully you now have a good basic understanding how to construct dictionaries. There's a lot more to go into here, but we will revisit dictionaries at later time. After this section all you need to know is how to create a dictionary and how to retrieve values from it.

Tuples

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

In this section, we will get a brief overview of the following:

```
1.) Constructing Tuples
2.) Basic Tuple Methods
3.) Immutability
4.) When to Use Tuples
```

You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

Constructing Tuples

The construction of a tuples use () with elements separated by commas. For example:

In [1]:
```
# Create a tuple
t = (1,2,3)
```

In [2]:
```
# Check len just like a list
len(t)
```

Out[2]:
```
3
```

In [3]:
```
# Can also mix object types
t = ('one',2)

# Show
t
```

Out[3]:
```
('one', 2)
```

In [4]:
```
# Use indexing just like we did in lists
t[0]
```

Out[4]:
```
'one'
```

In [5]:
```
# Slicing just like a list
t[-1]
```

Out[5]:
```
2
```

Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Let's look at two of them:

In [6]:
```
# Use .index to enter a value and return the index
t.index('one')
```

Out[6]:
```
0
```

In [7]:
```
# Use .count to count the number of times a value appears
t.count('one')
```

Out[7]:
```
1
```

Immutability

It can't be stressed enough that tuples are immutable. To drive that point home:

In [8]:
```
t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-1257c0aa9edd> in <module>()
----> 1 t[0]= 'change'

TypeError: 'tuple' object does not support item assignment
```

Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

In [9]:
```
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-b75f5b09ac19> in <module>()
----> 1 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

When to use Tuples

You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Up next Sets and Booleans!!

Set and Booleans

There are two other object types in Python that we should quickly cover: Sets and Booleans.

Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

In [1]:
```
x = set()
```

In [2]:
```
# We add to sets with the add() method
x.add(1)
```

In [3]:
```
#Show
x
```

Out[3]:
```
{1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

In [4]:
```
# Add a different element
x.add(2)
```

In [5]:
```
#Show
x
```

Out[5]:
```
{1, 2}
```

In [6]:
```
# Try to add the same element
x.add(1)
```

In [7]:
```
#Show
x
```

Out[7]:
```
{1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

In [8]:
```
# Create a list with repeats
list1 = [1,1,2,2,3,4,5,6,1,1]
```

In [9]:
```
# Cast as set to get unique values
set(list1)
```

Out[9]:
```
{1, 2, 3, 4, 5, 6}
```

Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None. Let's walk through a few quick examples of Booleans (we will dive deeper into them later in this course).

In [10]:
```
# Set object to be a boolean
a = True
```

In [11]:
```
#Show
a
```

Out[11]:
```
True
```

We can also use comparison operators to create booleans. We will go over all the comparison operators later on in the course.

In [12]:
```
# Output is boolean
1 > 2
```

Out[12]:
```
False
```

We can use None as a placeholder for an object that we don't want to reassign yet:

In [13]:
```
# None placeholder
b = None
```

In [14]:
```
# Show
print(b)
```

```
None
```

Thats it! You should now have a basic understanding of Python objects and data structure types. Next, go ahead and do the assessment test!

Files

Python uses file objects to interact with external files on your computer. These file objects can be any sort of file you have on your computer, whether it be an audio file, a text file, emails, Excel documents, etc. Note: You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available. (We will cover downloading modules later on in the course).

Python has a built-in open function that allows us to open and play with basic file types. First we will need a file though. We're going to use some IPython magic to create a text file!

IPython Writing a File

This function is specific to jupyter notebooks! Alternatively, quickly create a simple .txt file with sublime text editor.

In [1]:
```
%%writefile test.txt
Hello, this is a quick test file.
```

```
Overwriting test.txt
```

Python Opening a file

Let's being by opening the file test.txt that is located in the same directory as this notebook. For now we will work with files located in the same directory as the notebook or .py script you are using.

It is very easy to get an error on this step:

In [1]:
```
myfile = open('whoops.txt')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-1-dafe28ee473f> in <module>()
----> 1 myfile = open('whoops.txt')

FileNotFoundError: [Errno 2] No such file or directory: 'whoops.txt'
```

To avoid this error,make sure your .txt file is saved in the same location as your notebook, to check your notebook location, use **pwd**:

In [2]:
```
pwd
```

Out[2]:
```
'C:\\Users\\Marcial\\Pierian-Data-Courses\\Complete-Python-3-Bootcamp\\00-Python Object and Data Structure Basics'
```

**Alternatively, to grab files from any location on your computer, simply pass in the entire file path. **

For Windows you need to use double \ so python doesn't treat the second \ as an escape character, a file path is in the form:

```
myfile = open("C:\\Users\\YourUserName\\Home\\Folder\\myfile.txt")
```

For MacOS and Linux you use slashes in the opposite direction:

```
myfile = open("/Users/YouUserName/Folder/myfile.txt")
```

In [2]:
```
# Open the text.txt we made earlier
my_file = open('test.txt')
```

In [3]:
```
# We can now read the file
my_file.read()
```

Out[3]:
```
'Hello, this is a quick test file.'
```

In [4]:
```
# But what happens if we try to read it again?
my_file.read()
```

Out[4]:
```
''
```

This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So there is nothing left to read. We can reset the "cursor" like this:

In [5]:
```
# Seek to the start of file (index 0)
my_file.seek(0)
```

Out[5]:
```
0
```

In [6]:
```
# Now read again
my_file.read()
```

Out[6]:
```
'Hello, this is a quick test file.'
```

You can read a file line by line using the readlines method. Use caution with large files, since everything will be held in memory. We will learn how to iterate over large files later in the course.

In [7]:
```
# Readlines returns a list of the lines in the file
my_file.seek(0)
my_file.readlines()
```

Out[7]:
```
['Hello, this is a quick test file.']
```

When you have finished using a file, it is always good practice to close it.

In [8]:
```
my_file.close()
```

Writing to a File

By default, the `open()` function will only allow us to read the file. We need to pass the argument `'w'` to write over the file. For example:

In [9]:
```
# Add a second argument to the function, 'w' which stands for write.
# Passing 'w+' lets us read and write to the file

my_file = open('test.txt','w+')
```

Use caution!

Opening a file with `'w'` or `'w+'` truncates the original, meaning that anything that was in the original file **is deleted**!

In [10]:
```
# Write to the file
my_file.write('This is a new line')
```

Out[10]:
```
18
```

In [11]:
```
# Read the file
my_file.seek(0)
my_file.read()
```

Out[11]:
```
'This is a new line'
```

In [12]:
```
my_file.close()  # always do this when you're done with a file
```

Appending to a File

Passing the argument `'a'` opens the file and puts the pointer at the end, so anything written is appended. Like `'w+'`, `'a+'` lets us read and write to a file. If the file does not exist, one will be created.

In [13]:
```
my_file = open('test.txt','a+')
my_file.write('\nThis is text being appended to test.txt')
my_file.write('\nAnd another line here.')
```

Out[13]:
```
23
```

In [14]:
```
my_file.seek(0)
print(my_file.read())
```

```
This is a new line
This is text being appended to test.txt
And another line here.
```

In [15]:
```
my_file.close()
```

Appending with `%%writefile`

We can do the same thing using IPython cell magic:

In [16]:
```
%%writefile -a test.txt

This is text being appended to test.txt
And another line here.
```

```
Appending to test.txt
```

Add a blank space if you want the first line to begin on its own line, as Jupyter won't recognize escape sequences like `\n`

Iterating through a File

Lets get a quick preview of a for loop by iterating over a text file. First let's make a new text file with some IPython Magic:

In [17]:
```
%%writefile test.txt
First Line
Second Line
```

```
Overwriting test.txt
```

Now we can use a little bit of flow to tell the program to for through every line of the file and do something:

In [18]:
```
for line in open('test.txt'):
    print(line)
```

```
First Line

Second Line
```

Don't worry about fully understanding this yet, for loops are coming up soon. But we'll break down what we did above. We said that for every line in this text file, go ahead and print that line. It's important to note a few things here:

1. We could have called the "line" object anything (see example below).
2. By not calling `.read()` on the file, the whole text file was not stored in memory.
3. Notice the indent on the second line for print. This whitespace is required in Python.

In [19]:
```
# Pertaining to the first point above
for asdf in open('test.txt'):
    print(asdf)
```

```
First Line

Second Line
```

We'll learn a lot more about this later, but up next: Sets and Booleans!

Comparison Operators

In this lecture we will be learning about Comparison Operators in Python. These operators will allow us to compare variables and output a Boolean value (True or False).

If you have any sort of background in Math, these operators should be very straight forward.

First we'll present a table of the comparison operators and then work through some examples:

Table of Comparison Operators

In the table below, a=3 and b=4.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Let's now work through quick examples of each of these.

Equal

In [1]:
```
2 == 2
```

Out[1]:
```
True
```

In [2]:
```
1 == 0
```

Out[2]:
```
False
```

Note that `==` is a *comparison* operator, while `=` is an *assignment* operator.

Not Equal

In [3]:
```
2 != 1
```

Out[3]: True

In [4]:
```
2 != 2
```

Out[4]: False

Greater Than

In [5]:
```
2 > 1
```

Out[5]: True

In [6]:
```
2 > 4
```

Out[6]: False

Less Than

In [7]:
```
2 < 4
```

Out[7]: True

In [8]:
```
2 < 1
```

Out[8]: False

Greater Than or Equal to

In [9]:
```
2 >= 2
```

Out[9]: True

In [10]:
```
2 >= 1
```

Out[10]: True

Less than or Equal to

In [11]:
```
2 <= 2
```

Out[11]: True

In [12]:
```
2 <= 4
```

Out[12]:
```
True
```

Great! Go over each comparison operator to make sure you understand what each one is saying. But hopefully this was straightforward for you.

Next we will cover chained comparison operators

Chained Comparison Operators

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

In this lecture we will learn how to chain comparison operators and we will also introduce two other important statements in Python: **and** and **or**.

Let's look at a few examples of using chains:

In [1]:
```
1 < 2 < 3
```

Out[1]:
```
True
```

The above statement checks if 1 was less than 2 **and** if 2 was less than 3. We could have written this using an **and** statement in Python:

In [2]:
```
1<2 and 2<3
```

Out[2]:
```
True
```

The **and** is used to make sure two checks have to be true in order for the total check to be true. Let's see another example:

In [3]:
```
1 < 3 > 2
```

Out[3]:
```
True
```

The above checks if 3 is larger than both of the other numbers, so you could use **and** to rewrite it as:

In [4]:
```
1<3 and 3>2
```

Out[4]:
```
True
```

It's important to note that Python is checking both instances of the comparisons. We can also use **or** to write comparisons in Python. For example:

In [5]:
```
1==2 or 2<3
```

Out[5]:
```
True
```

Note how it was true; this is because with the **or** operator, we only need one *or* the other to be true. Let's see one more example to drive this home:

In [6]:
```
1==1 or 100==1
```

Out[6]:
```
True
```

Great! For an overview of this quick lesson: You should have a comfortable understanding of using **and** and **or** statements as well as reading chained comparison code.

Go ahead and go to the quiz for this section to check your understanding!

Introduction to Python Statements

In this lecture we will be doing a quick overview of Python Statements. This lecture will emphasize differences between Python and other languages such as C++.

There are two reasons we take this approach for learning the context of Python Statements:

```
1.) If you are coming from a different language this will rapidly accele
rate your understanding of Python.
2.) Learning about statements will allow you to be able to read other la
nguages more easily in the future.
```

Python vs Other Languages

Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"

Take a look at these two if statements (we will learn about building out if statements soon).

Version 1 (Other Languages)

```
if (a>b){
    a = 2;
    b = 4;
}
```

Version 2 (Python)

```
if a>b:
    a = 2
    b = 4
```

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

Python gets rid of () and {} by incorporating two main factors: a *colon* and *whitespace*. The statement is ended with a colon, and whitespace is used (indentation) to describe what takes place in case of the statement.

Another major difference is the lack of semicolons in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

Indentation

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

Other Languages

```
if (x)
    if(y)
        code-statement;
else
    another-code-statement;
```

Python

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

Note how Python is so heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

Now let's start diving deeper by coding these sort of statements in Python!

Time to code!

if, elif, else Statements

`if` Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with `elif` and `else` statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action."

Let's go ahead and look at the syntax format for `if` statements to get a better idea of this:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

First Example

Let's see a quick example of this:

In [1]:
```
if True:
    print('It was true!')
```

```
It was true!
```

Let's add in some else logic:

In [2]:
```
x = False

if x:
    print('x was True!')
else:
    print('I will be printed in any case where x is not true')
```

```
I will be printed in any case where x is not true
```

Multiple Branches

Let's get a fuller picture of how far `if` , `elif` , and `else` can take us!

We write this out in a nested structure. Take note of how the `if`, `elif`, and `else` line up in the code. This can help you see what `if` is related to what `elif` or `else` statements.

We'll reintroduce a comparison syntax for Python.

In [3]:
```
loc = 'Bank'

if loc == 'Auto Shop':
    print('Welcome to the Auto Shop!')
elif loc == 'Bank':
    print('Welcome to the bank!')
else:
    print('Where are you?')
```

```
Welcome to the bank!
```

Note how the nested `if` statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many `elif` statements as you want before you close off with an `else`.

Let's create two more simple examples for the `if`, `elif`, and `else` statements:

In [4]:
```
person = 'Sammy'

if person == 'Sammy':
    print('Welcome Sammy!')
else:
    print("Welcome, what's your name?")
```

```
Welcome Sammy!
```

In [5]:
```
person = 'George'

if person == 'Sammy':
    print('Welcome Sammy!')
elif person =='George':
    print('Welcome George!')
else:
    print("Welcome, what's your name?")
```

```
Welcome George!
```

Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

for Loops

A `for` loop acts as an iterator in Python; it goes through items that are in a *sequence* or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

We've already seen the `for` statement a little bit in past lectures but now let's formalize our understanding.

Here's the general format for a `for` loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use `if` statements to perform checks.

Let's go ahead and work through several example of `for` loops using a variety of data object types. We'll start simple and build more complexity later on.

Example 1

Iterating through a list

In [1]:
```
# We'll learn how to automate this sort of list in the next lecture
list1 = [1,2,3,4,5,6,7,8,9,10]
```

In [2]:
```
for num in list1:
    print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

Great! Hopefully this makes sense. Now let's add an `if` statement to check for even numbers. We'll first introduce a new concept here--the modulo.

Modulo

The modulo allows us to get the remainder in a division and uses the % symbol. For example:

In [3]:
```
17 % 5
```

Out[3]:
```
2
```

This makes sense since 17 divided by 5 is 3 remainder 2. Let's see a few more quick examples:

In [4]:
```
# 3 Remainder 1
10 % 3
```

Out[4]:
```
1
```

In [5]:
```
# 2 Remainder 4
18 % 7
```

Out[5]:
```
4
```

In [6]:
```
# 2 no remainder
4 % 2
```

Out[6]:
```
0
```

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an even number!

Back to the `for` loops!

Example 2

Let's print only the even numbers from that list!

In [7]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
```

```
2
4
6
8
10
```

We could have also put an `else` statement in there:

In [8]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

```
Odd number
2
Odd number
4
Odd number
6
Odd number
8
Odd number
10
```

Example 3

Another common idea during a `for` loop is keeping some sort of running tally during multiple loops. For example, let's create a `for` loop that sums up the list:

In [9]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum = list_sum + num

print(list_sum)
```

```
55
```

Great! Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a `+=` to perform the addition towards the sum. For example:

In [10]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum += num

print(list_sum)
```

```
55
```

Example 4

We've used `for` loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

In [11]:
```
for letter in 'This is a string.':
    print(letter)
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
.
```

Example 5

Let's now look at how a `for` loop can be used with a tuple:

In [12]:
```
tup = (1,2,3,4,5)

for t in tup:
    print(t)
```

```
1
2
3
4
5
```

Example 6

Tuples have a special quality when it comes to `for` loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the `for` loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

In [13]:
```
list2 = [(2,4),(6,8),(10,12)]
```

In [14]:
```
for tup in list2:
    print(tup)
```

```
(2, 4)
(6, 8)
(10, 12)
```

In [15]:
```
# Now with unpacking!
for (t1,t2) in list2:
    print(t1)
```

```
2
6
10
```

Cool! With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many objects will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

Example 7

In [16]:
```
d = {'k1':1,'k2':2,'k3':3}
```

In [17]:
```
for item in d:
    print(item)
```

```
k1
k2
k3
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

We're going to introduce three new Dictionary methods: **.keys()**, **.values()** and **.items()**

In Python each of these methods return a *dictionary view object*. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a view. Let's see it in action:

In [18]:
```
# Create a dictionary view object
d.items()
```

Out[18]:
```
dict_items([('k1', 1), ('k2', 2), ('k3', 3)])
```

Since the .items() method supports iteration, we can perform *dictionary unpacking* to separate keys and values just as we did in the previous examples.

In [19]:
```
# Dictionary unpacking
for k,v in d.items():
    print(k)
    print(v)
```

```
k1
1
k2
2
k3
3
```

If you want to obtain a true list of keys, values, or key/value tuples, you can *cast* the view as a list:

In [20]:
```
list(d.keys())
```

Out[20]:
```
['k1', 'k2', 'k3']
```

Remember that dictionaries are unordered, and that keys and values come back in arbitrary order. You can obtain a sorted list using sorted():

In [21]:
```
sorted(d.values())
```

Out[21]:
```
[1, 2, 3]
```

Conclusion

We've learned how to use for loops to iterate through tuples, lists, strings, and dictionaries. It will be an important tool for us, so make sure you know it well and understood the above examples.

More resources (http://www.tutorialspoint.com/python/python_for_loop.htm)

while Loops

The `while` statement in Python is one of most general ways to perform iteration. A `while` statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

Let's look at a few simple `while` loops in action.

In [1]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
```

Notice how many times the print statements occurred and how the `while` loop kept going until the True condition was met, which occurred once x==10. It's important to note that once this occurred the code stopped. Let's see how we could add an `else` statement:

In [2]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1

else:
    print('All Done!')
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
All Done!
```

break, continue, pass

We can use `break` , `continue` , and `pass` statements in our loops to add additional functionality for various cases. The three statements are defined by:

```
break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.
```

Thinking about `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

`break` and `continue` statements can appear anywhere inside the loop's body, but we will usually put them further nested in conjunction with an `if` statement to perform an action based on some condition.

Let's go ahead and look at some examples!

In [3]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x==3:
        print('x==3')
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
x==3
x is currently:  3
 x is still less than 10, adding 1 to x
continuing...
x is currently:  4
 x is still less than 10, adding 1 to x
continuing...
x is currently:  5
 x is still less than 10, adding 1 to x
continuing...
x is currently:  6
 x is still less than 10, adding 1 to x
continuing...
x is currently:  7
 x is still less than 10, adding 1 to x
continuing...
x is currently:  8
 x is still less than 10, adding 1 to x
continuing...
x is currently:  9
 x is still less than 10, adding 1 to x
continuing...
```

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

In [4]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x==3:
        print('Breaking because x==3')
        break
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
Breaking because x==3
```

Note how the other `else` statement wasn't reached and continuing was never printed!

After these brief but simple examples, you should feel comfortable using `while` statements in your code.

A word of caution however! It is possible to create an infinitely running loop with `while` statements. For example:

In [ ]:
```
# DO NOT RUN THIS CODE!!!!
while True:
    print("I'm stuck in an infinite loop!")
```

A quick note: If you *did* run the above cell, click on the Kernel menu above to restart the kernel!

Useful Operators

There are a few built-in functions and "operators" in Python that don't fit well into any category, so we will go over them in this lecture, let's begin!

range

The range function allows you to quickly *generate* a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Let's see some examples:

In [1]:
```
range(0,11)
```

Out[1]:
```
range(0, 11)
```

Note that this is a **generator** function, so to actually get a list out of it, we need to cast it to a list with **list()**. What is a generator? Its a special type of function that will generate information and not need to save it to memory. We haven't talked about functions or generators yet, so just keep this in your notes for now, we will discuss this in much more detail in later on in your training!

In [3]:
```
# Notice how 11 is not included, up to but not including 11, just like slice notc
list(range(0,11))
```

Out[3]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [4]:
```
list(range(0,12))
```

Out[4]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [6]:
```
# Third parameter is step size!
# step size just means how big of a jump/leap/step you
# take from the starting number to get to the next number.

list(range(0,11,2))
```

Out[6]:
```
[0, 2, 4, 6, 8, 10]
```

In [7]:
```
list(range(0,101,10))
```

Out[7]:
```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

enumerate

enumerate is a very useful function to use with for loops. Let's imagine the following situation:

In [8]:
```
index_count = 0

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

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

Keeping track of how many loops you've gone through is so common, that enumerate was created so you don't need to worry about creating and updating this index_count or loop_count variable

In [10]:
```
# Notice the tuple unpacking!

for i,letter in enumerate('abcde'):
    print("At index {} the letter is {}".format(i,letter))
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

zip

Notice the format enumerate actually returns, let's take a look by transforming it to a list()

In [12]:
```
list(enumerate('abcde'))
```

Out[12]:
```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

It was a list of tuples, meaning we could use tuple unpacking during our for loop. This data structure is actually very common in Python , especially when working with outside libraries. You can use the **zip()** function to quickly create a list of tuples by "zipping" up together two lists.

In [13]:
```
mylist1 = [1,2,3,4,5]
mylist2 = ['a','b','c','d','e']
```

In [15]:
```
# This one is also a generator! We will explain this later, but for now let's tr
zip(mylist1,mylist2)
```

Out[15]:
```
<zip at 0x1d205086f08>
```

In [17]:
```
list(zip(mylist1,mylist2))
```

Out[17]:
```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

To use the generator, we could just use a for loop

In [20]:
```
for item1, item2 in zip(mylist1,mylist2):
    print('For this tuple, first item was {} and second item was {}'.format(item
```

```
For this tuple, first item was 1 and second item was a
For this tuple, first item was 2 and second item was b
For this tuple, first item was 3 and second item was c
For this tuple, first item was 4 and second item was d
For this tuple, first item was 5 and second item was e
```

in operator

We've already seen the **in** keyword durng the for loop, but we can also use it to quickly check if an object is in a list

In [21]:
```
'x' in ['x','y','z']
```

Out[21]:
```
True
```

In [22]:
```
'x' in [1,2,3]
```

Out[22]:
```
False
```

min and max

Quickly check the minimum or maximum of a list with these functions.

In [26]:
```
mylist = [10,20,30,40,100]
```

In [27]:
```
min(mylist)
```

Out[27]:
```
10
```

In [44]:
```
max(mylist)
```

Out[44]:
```
100
```

random

Python comes with a built in random library. There are a lot of functions included in this random library, so we will only show you two useful functions for now.

In [29]:
```
from random import shuffle
```

In [35]:
```
# This shuffles the list "in-place" meaning it won't return
# anything, instead it will effect the list passed
shuffle(mylist)
```

In [36]:
```
mylist
```

Out[36]:
```
[40, 10, 100, 30, 20]
```

In [39]:
```
from random import randint
```

In [41]:
```
# Return random integer in range [a, b], including both end points.
randint(0,100)
```

Out[41]:
```
25
```

In [42]:
```
# Return random integer in range [a, b], including both end points.
randint(0,100)
```

Out[42]:
```
91
```

input

In [43]:
```
input('Enter Something into this box: ')
```

```
Enter Something into this box: great job!
```

Out[43]:
```
'great job!'
```

List Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.

List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line `for` loop built inside of brackets. For a simple example:

Example 1

In [1]:
```
# Grab every letter in string
lst = [x for x in 'word']
```

In [2]:
```
# Check
lst
```

Out[2]:
```
['w', 'o', 'r', 'd']
```

This is the basic idea of a list comprehension. If you're familiar with mathematical notation this format should feel familiar for example: x^2 : x in { 0,1,2...10 }

Let's see a few more examples of list comprehensions in Python:

Example 2

In [3]:
```
# Square numbers in range and turn into list
lst = [x**2 for x in range(0,11)]
```

In [4]:
```
lst
```

Out[4]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Example 3

Let's see how to add in `if` statements:

In [5]:
```
# Check for even numbers in a range
lst = [x for x in range(11) if x % 2 == 0]
```

In [6]:
```
lst
```

Out[6]:
```
[0, 2, 4, 6, 8, 10]
```

Example 4

Can also do more complicated arithmetic:

In [7]:
```
# Convert Celsius to Fahrenheit
celsius = [0,10,20.1,34.5]

fahrenheit = [((9/5)*temp + 32) for temp in celsius ]

fahrenheit
```

Out[7]:
```
[32.0, 50.0, 68.18, 94.1]
```

Example 5

We can also perform nested list comprehensions, for example:

In [8]:
```
lst = [ x**2 for x in [x**2 for x in range(11)]]
lst
```

Out[8]:
```
[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

Later on in the course we will learn about generator comprehensions. After this lecture you should feel comfortable reading and writing basic list comprehensions.

Statements Assessment Test

Let's test your knowledge!

Use `for` , .split(), and `if` to create a Statement that will print out words that start with 's':

In [ ]:
```
st = 'Print only the words that start with s in this sentence'
```

In [ ]:
```
#Code here
```

Use range() to print all the even numbers from 0 to 10.

In [ ]:
```
#Code Here
```

Use a List Comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.

In [ ]:
```
#Code in this cell
[]
```

Go through the string below and if the length of a word is even print "even!"

In [ ]:
```
st = 'Print every word in this sentence that has an even number of letters'
```

In [ ]:
```
#Code in this cell
```

Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

In [ ]:
```
#Code in this cell
```

Use List Comprehension to create a list of the first letters of every word in the string below:

In [ ]:
```
st = 'Create a list of the first letters of every word in this string'
```

In [ ]:
```
#Code in this cell
```

Great Job!

Statements Assessment Solutions

Use `for` , .split(), and `if` to create a Statement that will print out words that start with 's':

In [1]:
```
st = 'Print only the words that start with s in this sentence'
```

In [2]:
```
for word in st.split():
    if word[0] == 's':
        print(word)
```

```
start
s
sentence
```

Use range() to print all the even numbers from 0 to 10.

In [3]:
```
list(range(0,11,2))
```

Out[3]:
```
[0, 2, 4, 6, 8, 10]
```

Use List comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.

In [4]:
```
[x for x in range(1,51) if x%3 == 0]
```

Out[4]:
```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

Go through the string below and if the length of a word is even print "even!"

In [5]:
```
st = 'Print every word in this sentence that has an even number of letters'
```

In [6]:
```
for word in st.split():
    if len(word)%2 == 0:
        print(word+" <-- has an even length!")
```

```
word <-- has an even length!
in <-- has an even length!
this <-- has an even length!
sentence <-- has an even length!
that <-- has an even length!
an <-- has an even length!
even <-- has an even length!
number <-- has an even length!
of <-- has an even length!
```

Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

In [ ]:
```
for num in range(1,101):
    if num % 3 == 0 and num % 5 == 0:
        print("FizzBuzz")
    elif num % 3 == 0:
        print("Fizz")
    elif num % 5 == 0:
        print("Buzz")
    else:
        print(num)
```

Use a List Comprehension to create a list of the first letters of every word in the string below:

In [7]:
```
st = 'Create a list of the first letters of every word in this string'
```

In [8]:
```
[word[0] for word in st.split()]
```

Out[8]:
```
['C', 'a', 'l', 'o', 't', 'f', 'l', 'o', 'e', 'w', 'i', 't', 's']
```

Great Job!

Guessing Game Challenge

Let's use `while` loops to create a guessing game.

The Challenge:

Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

1. If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"
2. On a player's first turn, if their guess is
   - within 10 of the number, return "WARM!"
   - further than 10 away from the number, return "COLD!"
3. On all subsequent turns, if a guess is
   - closer to the number than the previous guess return "WARMER!"
   - farther from the number than the previous guess, return "COLDER!"
4. When the player's guess equals the number, tell them they've guessed correctly *and* how many guesses it took!

You can try this from scratch, or follow the steps outlined below. A separate Solution notebook has been provided. Good luck!

First, pick a random integer from 1 to 100 using the random module and assign it to a variable

Note: `random.randint(a,b)` returns a random integer in range `[a, b]`, including both end points.

In [ ]:

Next, print an introduction to the game and explain the rules

In [ ]:

Create a list to store guesses

Hint: zero is a good placeholder value. It's useful because it evaluates to "False"

In [ ]:

Write a `while` loop that asks for a valid guess. Test it a few times to make sure it works.

In [ ]:
```
while True:

    pass
```

Write a `while` loop that compares the player's guess to our number. If the player guesses correctly, break from the loop. Otherwise, tell the player if they're warmer or colder, and continue asking for guesses.

Some hints:

- it may help to sketch out all possible combinations on paper first!
- you can use the `abs()` function to find the positive difference between two numbers
- if you append all new guesses to the list, then the previous guess is given as `guesses[-2]`

In [ ]:
```
while True:

    # we can copy the code from above to take an input

    pass
```

That's it! You've just programmed your first game!

In the next section we'll learn how to turn some of these repetitive actions into *functions* that can be called whenever we need them.

Good Job!

Guessing Game Challenge - Solution

Let's use `while` loops to create a guessing game.

The Challenge:

Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

1. If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"
2. On a player's first turn, if their guess is
   - within 10 of the number, return "WARM!"
   - further than 10 away from the number, return "COLD!"
3. On all subsequent turns, if a guess is
   - closer to the number than the previous guess return "WARMER!"
   - farther from the number than the previous guess, return "COLDER!"
4. When the player's guess equals the number, tell them they've guessed correctly *and* how many guesses it took!

First, pick a random integer from 1 to 100 using the random module and assign it to a variable

Note: `random.randint(a,b)` returns a random integer in range `[a, b]`, including both end points.

In [1]:
```
import random

num = random.randint(1,100)
```

Next, print an introduction to the game and explain the rules

In [2]:
```
print("WELCOME TO GUESS ME!")
print("I'm thinking of a number between 1 and 100")
print("If your guess is more than 10 away from my number, I'll tell you you're CO
print("If your guess is within 10 of my number, I'll tell you you're WARM")
print("If your guess is farther than your most recent guess, I'll say you're gett
print("If your guess is closer than your most recent guess, I'll say you're getti
print("LET'S PLAY!")
```

```
WELCOME TO GUESS ME!
I'm thinking of a number between 1 and 100
If your guess is more than 10 away from my number, I'll tell you you're COLD
If your guess is within 10 of my number, I'll tell you you're WARM
If your guess is farther than your most recent guess, I'll say you're getting C
OLDER
If your guess is closer than your most recent guess, I'll say you're getting WA
RMER
LET'S PLAY!
```

Create a list to store guesses

Hint: zero is a good placeholder value. It's useful because it evaluates to "False"

In [3]:
```
guesses = [0]
```

Write a `while` loop that asks for a valid guess. Test it a few times to make sure it works.

In [4]:
```
while True:

    guess = int(input("I'm thinking of a number between 1 and 100.\n  What is you

    if guess < 1 or guess > 100:
        print('OUT OF BOUNDS! Please try again: ')
        continue

    break
```

```
I'm thinking of a number between 1 and 100.
  What is your guess? 500
OUT OF BOUNDS! Please try again:
I'm thinking of a number between 1 and 100.
  What is your guess? 50
```

Write a `while` loop that compares the player's guess to our number. If the player guesses correctly, break from the loop. Otherwise, tell the player if they're warmer or colder, and continue asking for guesses.

Some hints:

- it may help to sketch out all possible combinations on paper first!
- you can use the `abs()` function to find the positive difference between two numbers
- if you append all new guesses to the list, then the previous guess is given as `guesses[-2]`

In [5]:
```
while True:

    # we can copy the code from above to take an input
    guess = int(input("I'm thinking of a number between 1 and 100.\n  What is you

    if guess < 1 or guess > 100:
        print('OUT OF BOUNDS! Please try again: ')
        continue

    # here we compare the player's guess to our number
    if guess == num:
        print(f'CONGRATULATIONS, YOU GUESSED IT IN ONLY {len(guesses)} GUESSES!!
        break

    # if guess is incorrect, add guess to the list
    guesses.append(guess)

    # when testing the first guess, guesses[-2]==0, which evaluates to False
    # and brings us down to the second section

    if guesses[-2]:
        if abs(num-guess) < abs(num-guesses[-2]):
            print('WARMER!')
        else:
            print('COLDER!')

    else:
        if abs(num-guess) <= 10:
            print('WARM!')
        else:
            print('COLD!')
```

```
I'm thinking of a number between 1 and 100.
  What is your guess? 50
COLD!
I'm thinking of a number between 1 and 100.
  What is your guess? 75
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 85
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 92
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 80
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 78
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 82
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 83
COLDER!
I'm thinking of a number between 1 and 100.
```

```
  What is your guess? 81
CONGRATULATIONS, YOU GUESSED IT IN ONLY 9 GUESSES!!
```

That's it! You've just programmed your first game!

In the next section we'll learn how to turn some of these repetitive actions into *functions* that can be called whenever we need them.

Good Job!

Methods

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. Later on in the course we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.

Methods perform specific actions on an object and can also take arguments, just like a function. This lecture will serve as just a brief introduction to methods and get you thinking about overall design methods that we will touch back upon when we reach OOP in the course.

Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

You'll later see that we can think of methods as having an argument 'self' referring to the object itself. You can't see this argument but we will be using it later on in the course during the OOP lectures.

Let's take a quick look at what an example of the various methods a list has:

In [1]:
```
# Create a simple list
lst = [1,2,3,4,5]
```

Fortunately, with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. The methods for a list are:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

Let's try out a few of them:

append() allows us to add elements to the end of a list:

In [2]:
```
lst.append(6)
```

In [3]:
```
lst
```

Out[3]:
```
[1, 2, 3, 4, 5, 6]
```

Great! Now how about count()? The count() method will count the number of occurrences of an element in a list.

In [4]:
```
# Check how many times 2 shows up in the list
lst.count(2)
```

Out[4]:
```
1
```

You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function:

In [5]:
```
help(lst.count)
```

```
Help on built-in function count:

count(...) method of builtins.list instance
    L.count(value) -> integer -- return number of occurrences of value
```

Feel free to play around with the rest of the methods for a list. Later on in this section your quiz will involve using help and Google searching for methods of different types of objects!

Great! By this lecture you should feel comfortable calling methods of objects in Python!

Functions

Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

def Statements

Let's see how to build out a function's syntax in Python. It has the following form:

In [1]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a [built-in function in Python (https://docs.python.org/2/library/functions.html)](https://docs.python.org/2/library/functions.html) (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Example 1: A simple print 'hello' function

In [2]:
```
def say_hello():
    print('hello')
```

Call the function:

In [3]:
```
say_hello()
```

```
hello
```

Example 2: A simple greeting function

Let's write a function that greets people with their name.

In [4]:
```
def greeting(name):
    print('Hello %s' %(name))
```

In [5]:
```
greeting('Jose')
```

```
Hello Jose
```

Using return

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 3: Addition function

In [6]:
```
def add_num(num1,num2):
    return num1+num2
```

In [7]:
```
add_num(4,5)
```

Out[7]:
```
9
```

In [8]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [9]:
```
print(result)
```

```
9
```

What happens if we input two strings?

In [10]:
```
add_num('one','two')
```

Out[10]:
```
'onetwo'
```

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using `break`, `continue`, and `pass` statements in our code. We introduced these during the `while` lecture.

Finally let's go over a full example of creating a function to check if a number is prime (a common interview exercise).

We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

In [11]:
```
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print(num,'is not prime')
            break
    else: # If never mod zero, then prime
        print(num,'is prime!')
```

In [12]:
```
is_prime(16)
```

```
16 is not prime
```

In [13]:
```
is_prime(17)
```

```
17 is prime!
```

Note how the `else` lines up under `for` and not `if`. This is because we want the `for` loop to exhaust all possibilities in the range before printing our number is prime.

Also note how we break the code after the first print statement. As soon as we determine that a number is not prime we break out of the `for` loop.

We can actually improve this function by only checking to the square root of the target number, and by disregarding all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

In [14]:
```
import math

def is_prime2(num):
    '''
    Better method of checking for primes.
    '''
    if num % 2 == 0 and num > 2:
        return False
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True
```

In [15]:
```
is_prime2(18)
```

Out[15]:
```
False
```

Why don't we have any `break` statements? It should be noted that as soon as a function *returns* something, it shuts down. A function can deliver multiple print statements, but it will only obey one `return`.

Great! You should now have a basic understanding of creating your own functions to save yourself from repeatedly writing code!

Function Practice Exercises

Problems are arranged in increasing difficulty:

- Warmup - these can be solved using basic comparisons and methods
- Level 1 - these may involve if/then conditional statements and simple methods
- Level 2 - these may require iterating over sequences, usually with some kind of loop
- Challenging - these will take some creativity to solve

WARMUP SECTION:

LESSER OF TWO EVENS: Write a function that returns the lesser of two given numbers *if* both numbers are even, but returns the greater if one or both numbers are odd

```
lesser_of_two_evens(2,4) --> 2
lesser_of_two_evens(2,5) --> 5
```

In [ ]:
```
def lesser_of_two_evens(a,b):
    pass
```

In [ ]:
```
# Check
lesser_of_two_evens(2,4)
```

In [ ]:
```
# Check
lesser_of_two_evens(2,5)
```

ANIMAL CRACKERS: Write a function takes a two-word string and returns True if both words begin with same letter

```
animal_crackers('Levelheaded Llama') --> True
animal_crackers('Crazy Kangaroo') --> False
```

In [ ]:
```
def animal_crackers(text):
    pass
```

In [ ]:
```
# Check
animal_crackers('Levelheaded Llama')
```

In [ ]:
```
# Check
animal_crackers('Crazy Kangaroo')
```

MAKES TWENTY: Given two integers, return True if the sum of the integers is 20 *or* if one of the integers is 20. If not, return False

```
makes_twenty(20,10) --> True
makes_twenty(12,8) --> True
makes_twenty(2,3) --> False
```

In [ ]:
```
def makes_twenty(n1,n2):
    pass
```

In [ ]:
```
# Check
makes_twenty(20,10)
```

In [ ]:
```
# Check
makes_twenty(2,3)
```

LEVEL 1 PROBLEMS

OLD MACDONALD: Write a function that capitalizes the first and fourth letters of a name

```
old_macdonald('macdonald') --> MacDonald
```

Note: `'macdonald'.capitalize()` returns `'Macdonald'`

In [ ]:
```
def old_macdonald(name):
    pass
```

In [ ]:
```
# Check
old_macdonald('macdonald')
```

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

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

Note: The .join() method may be useful here. The .join() method allows you to join together strings in a list with some connector string. For example, some uses of the .join() method:

```
>>> "--".join(['a','b','c'])
>>> 'a--b--c'
```

This means if you had a list of words you wanted to turn back into a sentence, you could just join them with a single space string:

```
>>> " ".join(['Hello','world'])
>>> "Hello world"
```

In [ ]:
```
def master_yoda(text):
    pass
```

In [ ]:
```
# Check
master_yoda('I am home')
```

In [ ]:
```
# Check
master_yoda('We are ready')
```

ALMOST THERE: Given an integer n, return True if n is within 10 of either 100 or 200

```
almost_there(90) --> True
almost_there(104) --> True
almost_there(150) --> False
almost_there(209) --> True
```

NOTE: `abs(num)` returns the absolute value of a number

In [ ]:
```
def almost_there(n):
    pass
```

In [ ]:
```
# Check
almost_there(104)
```

In [ ]:
```
# Check
almost_there(150)
```

In [ ]:
```
# Check
almost_there(209)
```

LEVEL 2 PROBLEMS

FIND 33:

Given a list of ints, return True if the array contains a 3 next to a 3 somewhere.

```
has_33([1, 3, 3]) → True
has_33([1, 3, 1, 3]) → False
has_33([3, 1, 3]) → False
```

In [ ]:
```
def has_33(nums):
    pass
```

In [ ]:
```
# Check
has_33([1, 3, 3])
```

In [ ]:
```
# Check
has_33([1, 3, 1, 3])
```

In [ ]:
```
# Check
has_33([3, 1, 3])
```

PAPER DOLL: Given a string, return a string where for every character in the original there are three characters

```
paper_doll('Hello') --> 'HHHeeellllllooo'
paper_doll('Mississippi') --> 'MMMiiissssssiiipppppppiii'
```

In [ ]:
```
def paper_doll(text):
    pass
```

In [ ]:
```
# Check
paper_doll('Hello')
```

In [ ]:
```
# Check
paper_doll('Mississippi')
```

BLACKJACK: Given three integers between 1 and 11, if their sum is less than or equal to 21, return their sum. If their sum exceeds 21 *and* there's an eleven, reduce the total sum by 10. Finally, if the sum (even after adjustment) exceeds 21, return 'BUST'

```
blackjack(5,6,7) --> 18
blackjack(9,9,9) --> 'BUST'
blackjack(9,9,11) --> 19
```

In [ ]:
```
def blackjack(a,b,c):
    pass
```

In [ ]:
```
# Check
blackjack(5,6,7)
```

In [ ]:
```
# Check
blackjack(9,9,9)
```

In [ ]:
```
# Check
blackjack(9,9,11)
```

SUMMER OF '69: Return the sum of the numbers in the array, except ignore sections of numbers starting with a 6 and extending to the next 9 (every 6 will be followed by at least one 9). Return 0 for no numbers.

```
summer_69([1, 3, 5]) --> 9
summer_69([4, 5, 6, 7, 8, 9]) --> 9
summer_69([2, 1, 6, 9, 11]) --> 14
```

In [ ]:
```
def summer_69(arr):
    pass
```

In [ ]:
```
# Check
summer_69([1, 3, 5])
```

In [ ]:
```
# Check
summer_69([4, 5, 6, 7, 8, 9])
```

In [ ]:
```
# Check
summer_69([2, 1, 6, 9, 11])
```

CHALLENGING PROBLEMS

SPY GAME: Write a function that takes in a list of integers and returns True if it contains 007 in order

```
spy_game([1,2,4,0,0,7,5]) --> True
spy_game([1,0,2,4,0,5,7]) --> True
spy_game([1,7,2,0,4,5,0]) --> False
```

In [ ]:
```
def spy_game(nums):
    pass
```

In [ ]:
```
# Check
spy_game([1,2,4,0,0,7,5])
```

In [ ]:
```
# Check
spy_game([1,0,2,4,0,5,7])
```

In [ ]:
```
# Check
spy_game([1,7,2,0,4,5,0])
```

COUNT PRIMES: Write a function that returns the *number* of prime numbers that exist up to and including a given number

```
count_primes(100) --> 25
```

By convention, 0 and 1 are not prime.

In [ ]:
```
def count_primes(num):
    pass
```

In [ ]:
```
# Check
count_primes(100)
```

Just for fun:

PRINT BIG: Write a function that takes in a single letter, and returns a 5x5 representation of that letter

```
print_big('a')

out:   *
      * *
     *****
     *   *
     *   *
```

HINT: Consider making a dictionary of possible patterns, and mapping the alphabet to specific 5-line combinations of patterns.
For purposes of this exercise, it's ok if your dictionary stops at "E".

In [ ]:
```
def print_big(letter):
    pass
```

In [ ]:
```
print_big('a')
```

Great Job!

Function Practice Exercises - Solutions

Problems are arranged in increasing difficulty:

- Warmup - these can be solved using basic comparisons and methods
- Level 1 - these may involve if/then conditional statements and simple methods
- Level 2 - these may require iterating over sequences, usually with some kind of loop
- Challenging - these will take some creativity to solve

WARMUP SECTION:

LESSER OF TWO EVENS: Write a function that returns the lesser of two given numbers *if* both numbers are even, but returns the greater if one or both numbers are odd

```
lesser_of_two_evens(2,4) --> 2
lesser_of_two_evens(2,5) --> 5
```

In [1]:
```
def lesser_of_two_evens(a,b):
    if a%2 == 0 and b%2 == 0:
        return min(a,b)
    else:
        return max(a,b)
```

In [2]:
```
# Check
lesser_of_two_evens(2,4)
```

Out[2]: 2

In [3]:
```
# Check
lesser_of_two_evens(2,5)
```

Out[3]: 5

ANIMAL CRACKERS: Write a function takes a two-word string and returns True if both words begin with same letter

```
animal_crackers('Levelheaded Llama') --> True
animal_crackers('Crazy Kangaroo') --> False
```

In [4]:
```
def animal_crackers(text):
    wordlist = text.split()
    return wordlist[0][0] == wordlist[1][0]
```

In [5]:
```
# Check
animal_crackers('Levelheaded Llama')
```

Out[5]: True

In [6]:
```
# Check
animal_crackers('Crazy Kangaroo')
```

Out[6]: False

MAKES TWENTY: Given two integers, return True if the sum of the integers is 20 *or* if one of the integers is 20. If not, return False

```
makes_twenty(20,10) --> True
makes_twenty(12,8) --> True
makes_twenty(2,3) --> False
```

In [7]:
```
def makes_twenty(n1,n2):
    return (n1+n2)==20 or n1==20 or n2==20
```

In [8]:
```
# Check
makes_twenty(20,10)
```

Out[8]: True

In [9]:
```
# Check
makes_twenty(12,8)
```

Out[9]: True

In [10]:
```
#Check
makes_twenty(2,3)
```

Out[10]: False

LEVEL 1 PROBLEMS

OLD MACDONALD: Write a function that capitalizes the first and fourth letters of a name

```
old_macdonald('macdonald') --> MacDonald
```

Note: `'macdonald'.capitalize()` returns `'Macdonald'`

In [11]:
```
def old_macdonald(name):
    if len(name) > 3:
        return name[:3].capitalize() + name[3:].capitalize()
    else:
        return 'Name is too short!'
```

In [12]:
```
# Check
old_macdonald('macdonald')
```

Out[12]:
```
'MacDonald'
```

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

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

In [13]:
```
def master_yoda(text):
    return ' '.join(text.split()[::-1])
```

In [14]:
```
# Check
master_yoda('I am home')
```

Out[14]:
```
'home am I'
```

In [15]:
```
# Check
master_yoda('We are ready')
```

Out[15]:
```
'ready are We'
```

ALMOST THERE: Given an integer n, return True if n is within 10 of either 100 or 200

```
almost_there(90) --> True
almost_there(104) --> True
almost_there(150) --> False
almost_there(209) --> True
```

NOTE: `abs(num)` returns the absolute value of a number

In [16]:
```
def almost_there(n):
    return ((abs(100 - n) <= 10) or (abs(200 - n) <= 10))
```

In [17]:
```
# Check
almost_there(90)
```

Out[17]:
```
True
```

In [18]:
```
# Check
almost_there(104)
```

Out[18]:
```
True
```

In [19]:
```
# Check
almost_there(150)
```

Out[19]:
```
False
```

In [20]:
```
# Check
almost_there(209)
```

Out[20]: True

LEVEL 2 PROBLEMS

FIND 33:

Given a list of ints, return True if the array contains a 3 next to a 3 somewhere.

```
has_33([1, 3, 3]) → True
has_33([1, 3, 1, 3]) → False
has_33([3, 1, 3]) → False
```

In [21]:
```
def has_33(nums):
    for i in range(0, len(nums)-1):

        # nicer looking alternative in commented code
        #if nums[i] == 3 and nums[i+1] == 3:

        if nums[i:i+2] == [3,3]:
            return True

    return False
```

In [22]:
```
# Check
has_33([1, 3, 3])
```

Out[22]: True

In [23]:
```
# Check
has_33([1, 3, 1, 3])
```

Out[23]: False

In [24]:
```
# Check
has_33([3, 1, 3])
```

Out[24]: False

PAPER DOLL: Given a string, return a string where for every character in the original there are three characters

```
paper_doll('Hello') --> 'HHHeeellllllooo'
paper_doll('Mississippi') --> 'MMMiiissssssiiipppppppiii'
```

In [25]:
```
def paper_doll(text):
    result = ''
    for char in text:
        result += char * 3
    return result
```

In [26]:
```
# Check
paper_doll('Hello')
```

Out[26]: 'HHHeeellllllooo'

In [27]:
```
# Check
paper_doll('Mississippi')
```

Out[27]: 'MMMiiissssssiiissssssiiipppppppiii'

BLACKJACK: Given three integers between 1 and 11, if their sum is less than or equal to 21, return their sum. If their sum exceeds 21 *and* there's an eleven, reduce the total sum by 10. Finally, if the sum (even after adjustment) exceeds 21, return 'BUST'

```
blackjack(5,6,7) --> 18
blackjack(9,9,9) --> 'BUST'
blackjack(9,9,11) --> 19
```

In [28]:
```
def blackjack(a,b,c):

    if sum((a,b,c)) <= 21:
        return sum((a,b,c))
    elif sum((a,b,c)) <=31 and 11 in (a,b,c):
        return sum((a,b,c)) - 10
    else:
        return 'BUST'
```

In [29]:
```
# Check
blackjack(5,6,7)
```

Out[29]: 18

In [30]:
```
# Check
blackjack(9,9,9)
```

Out[30]: 'BUST'

In [31]:
```
# Check
blackjack(9,9,11)
```

Out[31]: 19

SUMMER OF '69: Return the sum of the numbers in the array, except ignore sections of numbers starting with a 6 and extending to the next 9 (every 6 will be followed by at least one 9). Return 0 for no numbers.

```
summer_69([1, 3, 5]) --> 9
summer_69([4, 5, 6, 7, 8, 9]) --> 9
summer_69([2, 1, 6, 9, 11]) --> 14
```

In [32]:
```
def summer_69(arr):
    total = 0
    add = True
    for num in arr:
        while add:
            if num != 6:
                total += num
                break
            else:
                add = False
        while not add:
            if num != 9:
                break
            else:
                add = True
                break
    return total
```

In [33]:
```
# Check
summer_69([1, 3, 5])
```

Out[33]: 9

In [34]:
```
# Check
summer_69([4, 5, 6, 7, 8, 9])
```

Out[34]: 9

In [35]:
```
# Check
summer_69([2, 1, 6, 9, 11])
```

Out[35]: 14

CHALLENGING PROBLEMS

SPY GAME: Write a function that takes in a list of integers and returns True if it contains 007 in order

```
spy_game([1,2,4,0,0,7,5]) --> True
spy_game([1,0,2,4,0,5,7]) --> True
spy_game([1,7,2,0,4,5,0]) --> False
```

In [36]:
```
def spy_game(nums):

    code = [0,0,7,'x']

    for num in nums:
        if num == code[0]:
            code.pop(0)   # code.remove(num) also works

    return len(code) == 1
```

In [37]:
```
# Check
spy_game([1,2,4,0,0,7,5])
```

Out[37]: True

In [38]:
```
# Check
spy_game([1,0,2,4,0,5,7])
```

Out[38]: True

In [39]:
```
# Check
spy_game([1,7,2,0,4,5,0])
```

Out[39]: False

COUNT PRIMES: Write a function that returns the *number* of prime numbers that exist up to and including a given number

```
count_primes(100) --> 25
```

By convention, 0 and 1 are not prime.

In [40]:
```
def count_primes(num):
    primes = [2]
    x = 3
    if num < 2:  # for the case of num = 0 or 1
        return 0
    while x <= num:
        for y in range(3,x,2):  # test all odd factors up to x-1
            if x%y == 0:
                x += 2
                break
        else:
            primes.append(x)
            x += 2
    print(primes)
    return len(primes)
```

In [41]:
```
# Check
count_primes(100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97]
```

Out[41]: 25

BONUS: Here's a faster version that makes use of the prime numbers we're collecting as we go!

In [42]:
```
def count_primes2(num):
    primes = [2]
    x = 3
    if num < 2:
        return 0
    while x <= num:
        for y in primes:  # use the primes list!
            if x%y == 0:
                x += 2
                break
        else:
            primes.append(x)
            x += 2
    print(primes)
    return len(primes)
```

In [43]:
```
count_primes2(100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97]
```

Out[43]: 25

Just for fun, not a real problem :)

PRINT BIG: Write a function that takes in a single letter, and returns a 5x5 representation of that letter

```
print_big('a')

out:   *
      * *
     *****
     *   *
     *   *
```

HINT: Consider making a dictionary of possible patterns, and mapping the alphabet to specific 5-line combinations of patterns.
For purposes of this exercise, it's ok if your dictionary stops at "E".

In [44]:
```
def print_big(letter):
    patterns = {1:'  *  ',2:' * * ',3:'*   *',4:'*****',5:'**** ',6:'   * ',7:' 
    alphabet = {'A':[1,2,4,3,3],'B':[5,3,5,3,5],'C':[4,9,9,9,4],'D':[5,3,3,3,5],
    for pattern in alphabet[letter.upper()]:
        print(patterns[pattern])
```

In [45]:
```
print_big('a')
```

```
  *
 * *
*****
*   *
*   *
```

Great Job!

Lambda Expressions, Map, and Filter

Now its time to quickly learn about two built in functions, filter and map. Once we learn about how these operate, we can learn about the lambda expression, which will come in handy when you begin to develop your skills further!

map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

In [1]:
```
def square(num):
    return num**2
```

In [2]:
```
my_nums = [1,2,3,4,5]
```

In [5]:
```
map(square,my_nums)
```

Out[5]:
```
<map at 0x205baec21d0>
```

In [7]:
```
# To get the results, either iterate through map()
# or just cast to a list
list(map(square,my_nums))
```

Out[7]:
```
[1, 4, 9, 16, 25]
```

The functions can also be more complex

In [8]:
```
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'even'
    else:
        return mystring[0]
```

In [9]:
```
mynames = ['John','Cindy','Sarah','Kelly','Mike']
```

In [10]:
```
list(map(splicer,mynames))
```

Out[10]:
```
['even', 'C', 'S', 'K', 'even']
```

filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

In [12]:
```
def check_even(num):
    return num % 2 == 0
```

In [13]:
```
nums = [0,1,2,3,4,5,6,7,8,9,10]
```

In [15]:
```
filter(check_even,nums)
```

Out[15]:
```
<filter at 0x205baed4710>
```

In [16]:
```
list(filter(check_even,nums))
```

Out[16]:
```
[0, 2, 4, 6, 8, 10]
```

lambda expression

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

lambda's body is a single expression, not a block of statements.

- The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

Lets slowly break down a lambda expression by deconstructing a function:

In [17]:
```
def square(num):
    result = num**2
    return result
```

In [18]:
```
square(2)
```

Out[18]:
```
4
```

We could simplify it:

In [19]:
```
def square(num):
    return num**2
```

In [20]:
```
square(2)
```

Out[20]:
```
4
```

We could actually even write this all on one line.

In [21]:
```
def square(num): return num**2
```

In [22]:
```
square(2)
```

Out[22]:
```
4
```

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

In [23]:
```
lambda num: num ** 2
```

Out[23]:
```
<function __main__.<lambda>>
```

In [25]:
```
# You wouldn't usually assign a name to a lambda expression, this is just for der
square = lambda num: num **2
```

In [26]:
```
square(2)
```

Out[26]:
```
4
```

So why would use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression. Let's repeat some of the examples from above with a lambda expression

In [29]:
```
list(map(lambda num: num ** 2, my_nums))
```

Out[29]:
```
[1, 4, 9, 16, 25]
```

In [30]:
```
list(filter(lambda n: n % 2 == 0,nums))
```

Out[30]:
```
[0, 2, 4, 6, 8, 10]
```

Here are a few more examples, keep in mind the more comples a function is, the harder it is to translate into a lambda expression, meaning sometimes its just easier (and often the only way) to create the def keyword function.

** Lambda expression for grabbing the first character of a string: **

In [31]:
```
lambda s: s[0]
```

Out[31]:
```
<function __main__.<lambda>>
```

** Lambda expression for reversing a string: **

In [32]:
```
lambda s: s[::-1]
```

Out[32]:
```
<function __main__.<lambda>>
```

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

In [34]:
```
lambda x,y : x + y
```

Out[34]:
```
<function __main__.<lambda>>
```

You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

Nested Statements and Scope

Now that we have gone over writing our own functions, it's important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

Let's start with a quick thought experiment; imagine the following code:

In [1]:
```
x = 25

def printer():
    x = 50
    return x

# print(x)
# print(printer())
```

What do you imagine the output of printer() is? 25 or 50? What is the output of print x? 25 or 50?

In [2]:
```
print(x)
```

```
25
```

In [3]:
```
print(printer())
```

```
50
```

Interesting! But how does Python know which **x** you're referring to in your code? This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as **x** in this case) you are referencing in your code. Lets break down the rules:

This idea of scope in your code is very important to understand in order to properly assign and call variable names.

In simple terms, the idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.
2. Name references search (at most) four scopes, these are:
    - local
    - enclosing functions
    - global
    - built-in
3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

The statement in #2 above can be defined by the LEGB rule.

LEGB Rule:

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

Quick examples of LEGB

Local

In [4]:
```
# x is local here:
f = lambda x:x**2
```

Enclosing function locals

This occurs when we have a function inside a function (nested functions)

In [5]:
```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

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

    hello()

greet()
```

```
Hello Sammy
```

Note how Sammy was used, because the hello() function was enclosed inside of the greet function!

Global

Luckily in Jupyter a quick way to test for global variables is to see if another cell recognizes the variable!

In [6]:
```
print(name)
```

```
This is a global name
```

Built-in

These are the built-in function names in Python (don't overwrite these!)

In [7]:
```
len
```

Out[7]:
```
<function len>
```

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example:

In [8]:
```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

```
x is 50
Changed local x to 2
x is still 50
```

The first time that we print the value of the name **x** with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to **x*. *The name* *x** is local to our function. So, when we change the value of **x** in the function, the **x** defined in the main block remains unaffected.

With the last print statement, we display the value of **x** as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The `global` statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the `global` statement. It is impossible to assign a value to a

variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block.

Example:

In [9]:
```
x = 50

def func():
    global x
    print('This function is now using the global x!')
    print('Because of global x is: ', x)
    x = 2
    print('Ran func(), changed global x to', x)

print('Before calling func(), x is: ', x)
func()
print('Value of x (outside of func()) is: ', x)
```

```
Before calling func(), x is:  50
This function is now using the global x!
Because of global x is:  50
Ran func(), changed global x to 2
Value of x (outside of func()) is:  2
```

The `global` statement is used to declare that **x** is a global variable - hence, when we assign a value to **x** inside the function, that change is reflected when we use the value of **x** in the main block.

You can specify more than one global variable using the same global statement e.g. `global x, y, z`.

Conclusion

You should now have a good understanding of Scope (you may have already intuitively felt right about Scope which is great!) One last mention is that you can use the **globals()** and **locals()** functions to check what are your current local and global variables.

Another thing to keep in mind is that everything in Python is an object! I can assign variables to functions just like I can with numbers! We will go over this again in the decorator section of the course!

*args and **kwargs ¶

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

In [1]:
```
def myfunc(a,b):
    return sum((a,b))*.05

myfunc(40,60)
```

Out[1]:
```
5.0
```

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

In [2]:
```
def myfunc(a=0,b=0,c=0,d=0,e=0):
    return sum((a,b,c,d,e))*.05

myfunc(40,60,20)
```

Out[2]:
```
6.0
```

Obviously this is not a very efficient solution, and that's where `*args` comes in.

*args

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

In [3]:
```
def myfunc(*args):
    return sum(args)*.05

myfunc(40,60,20)
```

Out[3]:
```
6.0
```

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

In [4]:
```
def myfunc(*spam):
    return sum(spam)*.05

myfunc(40,60,20)
```

Out[4]:
```
6.0
```

**kwargs

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

In [5]:
```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")  # review String Format
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

In [6]:
```
myfunc()
```

```
I don't like fruit
```

*args and **kwargs combined

You can pass `*args` and `**kwargs` into the same function, but `*args` have to appear before `**kwargs`

In [7]:
```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fru
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

In [8]:
```
myfunc(fruit='cherries',juice='orange','eggs','spam')
```

```
  File "<ipython-input-8-fc6ff65addcc>", line 1
    myfunc(fruit='cherries',juice='orange','eggs','spam')
                                           ^
SyntaxError: positional argument follows keyword argument
```

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.

That's it! Now you should understand how `*args` and `**kwargs` provide the flexibilty to work with arbitrary numbers of arguments!

Functions and Methods Homework

Complete the following questions:

Write a function that computes the volume of a sphere given its radius.

The volume of a sphere is given as

$$\frac{4}{3}\pi r^3$$

In [1]:
```
def vol(rad):
    pass
```

In [2]:
```
# Check
vol(2)
```

Out[2]:
```
33.49333333333333
```

Write a function that checks whether a number is in a given range (inclusive of high and low)

In [3]:
```
def ran_check(num,low,high):
    pass
```

In [4]:
```
# Check
ran_check(5,2,7)
```

```
5 is in the range between 2 and 7
```

If you only wanted to return a boolean:

In [5]:
```
def ran_bool(num,low,high):
    pass
```

In [6]:
```
ran_bool(3,1,10)
```

Out[6]:
```
True
```

Write a Python function that accepts a string and calculates the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

HINT: Two string methods that might prove useful: **.isupper()** and **.islower()**

If you feel ambitious, explore the Collections module to solve this problem!

In [7]:
```
def up_low(s):
    pass
```

In [8]:
```
s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
up_low(s)
```

```
Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]
```

In [9]:
```
def unique_list(lst):
    pass
```

In [10]:
```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

Out[10]:
```
[1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

In [11]:
```
def multiply(numbers):
    pass
```

In [12]:
```
multiply([1,2,3,-4])
```

Out[12]:
```
-24
```

Write a Python function that checks whether a passed in string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

In [13]:
```
def palindrome(s):
    pass
```

In [14]:
```
palindrome('helleh')
```

Out[14]:
```
True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the al
phabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

In [15]:
```
import string

def ispangram(str1, alphabet=string.ascii_lowercase):
    pass
```

In [16]:
```
ispangram("The quick brown fox jumps over the lazy dog")
```

Out[16]:
```
True
```

In [17]:
```
string.ascii_lowercase
```

Out[17]:
```
'abcdefghijklmnopqrstuvwxyz'
```

Great Job!

Functions and Methods Homework Solutions

Write a function that computes the volume of a sphere given its radius.

In [1]:
```
def vol(rad):
    return (4/3)*(3.14)*(rad**3)
```

In [2]:
```
# Check
vol(2)
```

Out[2]:
```
33.49333333333333
```

Write a function that checks whether a number is in a given range (inclusive of high and low)

In [3]:
```
def ran_check(num,low,high):
    #Check if num is between low and high (including low and high)
    if num in range(low,high+1):
        print('{} is in the range between {} and {}'.format(num,low,high))
    else:
        print('The number is outside the range.')
```

In [4]:
```
# Check
ran_check(5,2,7)
```

```
5 is in the range between 2 and 7
```

If you only wanted to return a boolean:

In [5]:
```
def ran_bool(num,low,high):
    return num in range(low,high+1)
```

In [6]:
```
ran_bool(3,1,10)
```

Out[6]:
```
True
```

Write a Python function that accepts a string and calculates the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

If you feel ambitious, explore the Collections module to solve this problem!

In [7]:
```
def up_low(s):
    d={"upper":0, "lower":0}
    for c in s:
        if c.isupper():
            d["upper"]+=1
        elif c.islower():
            d["lower"]+=1
        else:
            pass
    print("Original String : ", s)
    print("No. of Upper case characters : ", d["upper"])
    print("No. of Lower case Characters : ", d["lower"])
```

In [8]:
```
s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
up_low(s)
```

```
Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]
```

In [9]:
```
def unique_list(lst):
    # Also possible to use list(set())
    x = []
    for a in lst:
        if a not in x:
            x.append(a)
    return x
```

In [10]:
```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

Out[10]:
```
[1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

In [11]:
```
def multiply(numbers):
    total = 1
    for x in numbers:
        total *= x
    return total
```

In [12]:
```
multiply([1,2,3,-4])
```

Out[12]:
```
-24
```

Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

In [13]:
```
def palindrome(s):

    s = s.replace(' ','') # This replaces all spaces ' ' with no space ''. (Fixes
    return s == s[::-1]   # Check through slicing
```

In [14]:
```
palindrome('nurses run')
```

Out[14]:
```
True
```

In [15]:
```
palindrome('abcba')
```

Out[15]:
```
True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the al
phabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

In [16]:
```
import string

def ispangram(str1, alphabet=string.ascii_lowercase):
    alphaset = set(alphabet)
    return alphaset <= set(str1.lower())
```

In [17]:
```
ispangram("The quick brown fox jumps over the lazy dog")
```

Out[17]:
```
True
```

In [18]:
```
string.ascii_lowercase
```

Out[18]:
```
'abcdefghijklmnopqrstuvwxyz'
```

Milestone Project 1

Congratulations on making it to your first milestone!

You've already learned a ton and are ready to work on a real project.

Your assignment: Create a Tic Tac Toe game. You are free to use any IDE you like.

Here are the requirements:

- 2 players should be able to play the game (both sitting at the same computer)
- The board should be printed out every time a player makes a move
- You should be able to accept input of the player position and then place a symbol on the board

Feel free to use Google to help you figure anything out (but don't just Google "Tic Tac Toe in Python" otherwise you won't learn anything!) Keep in mind that this project can take anywhere between several hours to several days.

There are 4 Jupyter Notebooks related to this assignment:

- This Assignment Notebook
- A "Walkthrough Steps Workbook" Notebook
- A "Complete Walkthrough Solution" Notebook
- An "Advanced Solution" Notebook

I encourage you to just try to start the project on your own without referencing any of the notebooks. If you get stuck, check out the next lecture which is a text lecture with helpful hints and steps. If you're still stuck after that, then check out the Walkthrough Steps Workbook, which breaks up the project in steps for you to solve. Still stuck? Then check out the Complete Walkthrough Solution video for more help on approaching the project!

There are parts of this that will be a struggle...and that is good! I have complete faith that if you have made it this far through the course you have all the tools and knowledge to tackle this project. Remember, it's totally open book, so take your time, do a little research, and remember:

HAVE FUN!