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

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

```
Out[1]:  3
```

```python
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.

```python
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.

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

Out[12]: 10

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

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

```python
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:

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

Out[15]: 10

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

```python
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]:
```python
# 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]:
```python
"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]:
```python
# We can simply declare a string
'Hello World'
```

Out[6]: `'Hello World'`

In [7]:
```python
# 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]:
```python
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!

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

```python
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:

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

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

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

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

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

In [6]:
```python
# 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]:
```python
my_dict['key1']
```

Out[7]: 123

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

In [9]:
```python
#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]:
```python
# 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]:
```python
# Create a new dictionary
d = {}
```

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

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

In [14]:
```python
#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]:
```python
# 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]:
```python
# 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]:
```python
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}
```

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

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

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

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

In [20]:
```python
# 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]:
```python
# We add to sets with the add() method
x.add(1)
```

In [3]:
```python
#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]:
```python
# Add a different element
x.add(2)
```

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

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

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

In [7]:
```python
#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]:
```python
# 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]:
```python
# Open the text.txt we made earlier
my_file = open('test.txt')
```

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

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

In [4]:
```python
# 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]:
```python
# Seek to the start of file (index 0)
my_file.seek(0)
```

Out[5]: 0

In [6]:
```python
# 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]:
```python
# 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]:
```python
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!