Numbers

August 20, 2017

1 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 Python 2 vs 3 in division
4.) Object Assignment in Python
```

1.1 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:

Numbers in Python
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.

1.1.1 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
```

1.1.2 Python 3 Alert!

Whoa! What just happened? Last time I checked, 3 divided by 2 is equal 1.5 not 1!

The reason we get this result is because we are using Python 2. In Python 2, the / symbol performs what is known as "*classic*" division, this means that the decimal points are truncated (cut off). In Python 3 however, a single / performs "*true*" division. So you would get 1.5 if you had inputed 3/2 in Python 3.

So what do we do if we are using Python 2 to avoid this?

There are two options:

Specify one of the numbers to be a float:

```
In [11]: # Specifying one of the numbers as a float
         3.0/2

Out[11]: 1.5

In [12]: # Works for either number
         3/2.0

Out[12]: 1.5
```

We could also "cast" the type using a function that basically turns integers into floats. This function, unsurprisingly, is called float().

```
In [14]: # We can use this float() function to cast integers as floats:
         float(3)/2

Out[14]: 1.5
```

We will go over functions in much more detail later on in this course, so don't worry if you are confused by the syntax here. Consider this a sneak preview.

One more "sneak preview" we can use to deal with classic division in Python 2 is importing from a module called **future**.

This is a module in Python 2 that has Python 3 functions, this basically allows you to import Python 3 functions into Python 2. We will go over imports and modules later in the course, so don't worry about fully understanding the import statement right now!

```
In [15]: from __future__ import division
         3/2

Out[15]: 1.5
```

When you import division from the **future** you won't need to worry about classic division occurring anymore anywhere in your code!

1.1.3 Arithmetic continued

```
In [16]: # Powers
         2**3

Out[16]: 8

In [17]: # Can also do roots this way
         4**0.5

Out[17]: 2.0

In [18]: # Order of Operations followed in Python
         2 + 10 * 10 + 3

Out[18]: 105

In [19]: # Can use parenthesis to specify orders
         (2+10) * (10+3)

Out[19]: 156
```

1.2 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 [37]: # 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 [38]: # Adding the objects
         a+a

Out[38]: 10
```

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

```
In [39]: # Reassignment
         a = 10
```

```
In [40]: # Check
         a

Out[40]: 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 [41]: # Check
         a

Out[41]: 10

In [42]: # Use A to redefine A
         a = a + a

In [43]: # Check
         a

Out[43]: 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 :'",<>/?|\()!@#$%^&*~-+
3. It's considered best practice (PEP8) that the names are lowercase.
```

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

```
In [44]: # 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 [46]: # Show my taxes!
         my_taxes

Out[46]: 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!

Strings

August 21, 2017

1 Strings

Strings are used in Python to record text information, such as name. 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.) Differences in Printing in Python 2 vs 3
4.) String Indexing and Slicing
5.) String Properties
6.) String Methods
7.) Print Formatting
```

1.1 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 will create an error'


          File "<ipython-input-4-6565b0b7b5e3>", line 2
        ' I'm using single quotes, but 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 [10]: "Now I'm ready to use the single quotes inside a string!"

Out[10]: "Now I'm ready to use the single quotes inside a string!"
```

Now let's learn about printing strings!

1.2 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 [11]: # We can simply declare a string
         'Hello World'

Out[11]: 'Hello World'

In [12]: # note that we can't output multiple strings this way
         'Hello World 1'
         'Hello World 2'

Out[12]: 'Hello World 2'
```

We can use a print statement to print a string.

```
In [13]: 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?
```

1.2.1 Python 3 Alert!

Something to note. In Python 3, print is a function, not a statement. So you would print statements like this: print('Hello World')

If you want to use this functionality in Python2, you can import form the **future** module.

A word of caution, after importing this you won't be able to choose the print statement method anymore. So pick whichever one you prefer depending on your Python installation and continue on with it.

```
In [32]: # To use print function from Python 3 in Python 2
         from __future__ import print_function

         print('Hello World')
```

```
Hello World
```

1.3 String Basics

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

```
In [33]: len('Hello World')
```

```
Out[33]: 11
```

1.4 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 the walk through a few examples of indexing.

```
In [1]: # Assign s as a string
        s = 'Hello World'
```

```
In [35]: #Check
         s
```

```
Out[35]: 'Hello World'
```

```
In [36]: # Print the object
         print(s)
```

```
Hello World
```

Let's start indexing!

```
In [21]: # Show first element (in this case a letter)
         s[0]
```

```
Out[21]: 'H'

In [22]: s[1]

Out[22]: 'e'

In [23]: s[2]

Out[23]: 'l'
```

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

```
In [24]: # Grab everything past the first term all the way to the length of s which
         s[1:]

Out[24]: 'ello World'

In [25]: # Note that there is no change to the original s
         s

Out[25]: 'Hello World'

In [26]: # Grab everything UP TO the 3rd index
         s[:3]

Out[26]: '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 [27]: #Everything
         s[:]

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

We can also use negative indexing to go backwards.

```
In [28]: # Last letter (one index behind 0 so it loops back around)
         s[-1]

Out[28]: 'd'

In [29]: # Grab everything but the last letter
         s[:-1]

Out[29]: '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 [42]: # Grab everything, but go in steps size of 1
         s[::1]

Out[42]: 'Hello World'

In [46]: # Grab everything, but go in step sizes of 2
         s[::2]

Out[46]: 'HloWrd'

In [47]: # We can use this to print a string backwards
         s[::-1]

Out[47]: 'dlroW olleH'
```

1.5 String Properties

Its 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 [48]: s

Out[48]: 'Hello World'

In [49]: # Let's try to change the first letter to 'x'
         s[0] = 'x'


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-49-3a9c668aa5ab> 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 [50]: s

Out[50]: 'Hello World'

In [52]: # Concatenate strings!
         s + ' concatenate me!'
```

```
Out[52]: 'Hello World concatenate me!'

In [53]: # We can reassign s completely though!
         s = s + ' concatenate me!'

In [54]: print(s)

Hello World concatenate me!


In [58]: s

Out[58]: 'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

```
In [59]: letter = 'z'

In [60]: letter*10

Out[60]: 'zzzzzzzzzz'
```

1.6 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 [61]: s

Out[61]: 'Hello World concatenate me!'

In [62]: # Upper Case a string
         s.upper()

Out[62]: 'HELLO WORLD CONCATENATE ME!'

In [65]: # Lower case
         s.lower()

Out[65]: 'hello world concatenate me!'

In [67]: # Split a string by blank space (this is the default)
         s.split()

Out[67]: ['Hello', 'World', 'concatenate', 'me!']

In [68]: # Split by a specific element (doesn't include the element that was split
         s.split('W')

Out[68]: ['Hello ', 'orld concatenate me!']
```

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

1.7 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 [79]: 'Insert another string with curly brackets: {}'.format('The inserted strin
```

```
Out[79]: '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!

1.8 Next up: Lists!

```
In [ ]:
```

Print Formatting

August 21, 2017

1 Print Formatting

In this lecture we will briefly cover the various ways to format your print statements. As you code more and more, you will probably want to have print statements that can take in a variable into a printed string statement.

The most basic example of a print statement is:

```
In [17]: print 'This is a string'

This is a string
```

1.1 Strings

You can use the %s to format strings into your print statements.

```
In [4]: s = 'STRING'
        print 'Place another string with a mod and s: %s' %(s)

Place another string with a mod and s: STRING
```

1.2 Floating Point Numbers

Floating point numbers use the format %n1.n2f where the n1 is the total minimum number of digits the string should contain (these may be filled with whitespace if the entire number does not have this many digits. The n2 placeholder stands for how many numbers to show past the decimal point. Lets see some examples:

```
In [12]: print 'Floating point numbers: %1.2f' %(13.144)

Floating point numbers: 13.14


In [13]: print 'Floating point numbers: %1.0f' %(13.144)

Floating point numbers: 13
```

```
In [14]: print 'Floating point numbers: %1.5f' %(13.144)

Floating point numbers: 13.14400


In [15]: print 'Floating point numbers: %10.2f' %(13.144)

Floating point numbers:      13.14


In [19]: print 'Floating point numbers: %25.2f' %(13.144)

Floating point numbers:                     13.14
```

1.3 Conversion Format methods.

It should be noted that two methods %s and %r actually convert any python object to a string using two separate methods: str() and repr(). We will learn more about these functions later on in the course, but you should note you can actually pass almost any Python object with these two methods and it will work:

```
In [23]: print 'Here is a number: %s. Here is a string: %s' %(123.1,'hi')

Here is a number: 123.1. Here is a string: hi


In [24]: print 'Here is a number: %r. Here is a string: %r' %(123.1,'hi')

Here is a number: 123.1. Here is a string: 'hi'
```

1.4 Multiple Formatting

Pass a tuple to the modulo symbol to place multiple formats in your print statements:

```
In [22]: print 'First: %s, Second: %1.2f, Third: %r' %('hi!',3.14,22)

First: hi!, Second: 3.14, Third: 22
```

2 Using the string .format() method

The best way to format objects into your strings for print statements is using the format method. The syntax is:

```
'String here {var1} then also {var2}'.format(var1='something1',var2='something2')
```

Lets see some examples:

```
In [27]: print 'This is a string with an {p}'.format(p='insert')

This is a string with an insert


In [28]: # Multiple times:
         print 'One: {p}, Two: {p}, Three: {p}'.format(p='Hi!')

One: Hi!, Two: Hi!, Three: Hi!


In [29]: # Several Objects:
         print 'Object 1: {a}, Object 2: {b}, Object 3: {c}'.format(a=1,b='two',c=1

Object 1: 1, Object 2: two, Object 3: 12.3
```

That is the basics of string formatting! Remember that Python 3 uses a print() function, not the print statement!

Lists

March 17, 2017

1 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 [4]: 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 [6]: len(my_list)

Out[6]: 4
```

1.0.1 Indexing and Slicing

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

```
In [7]: my_list = ['one','two','three',4,5]

In [10]: # Grab element at index 0
         my_list[0]
```

```
Out[10]: 'one'

In [11]: # Grab index 1 and everything past it
         my_list[1:]

Out[11]: ['two', 'three', 4, 5]

In [13]: # Grab everything UP TO index 3
         my_list[:3]

Out[13]: ['one', 'two', 'three']
```

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

```
In [14]: my_list + ['new item']

Out[14]: ['one', 'two', 'three', 4, 5, 'new item']
```

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

```
In [15]: my_list

Out[15]: ['one', 'two', 'three', 4, 5]
```

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

```
In [16]: # Reassign
         my_list = my_list + ['add new item permanently']

In [18]: my_list

Out[18]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

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

```
In [20]: # Make the list double
         my_list * 2

Out[20]: ['one',
          'two',
          'three',
          4,
          5,
          'add new item permanently',
          'one',
          'two',
          'three',
          4,
          5,
          'add new item permanently']

In [23]: # Again doubling not permanent
         my_list

Out[23]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

1.1 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 [42]: # Create a new list
         l = [1,2,3]
```

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

```
In [43]: # Append
         l.append('append me!')

In [44]: # Show
         l

Out[44]: [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 [45]: # Pop off the 0 indexed item
         l.pop(0)

Out[45]: 1

In [46]: # Show
         l

Out[46]: [2, 3, 'append me!']

In [47]: # Assign the popped element, remember default popped index is -1
         popped_item = l.pop()

In [48]: popped_item

Out[48]: 'append me!'

In [49]: # Show remaining list
         l

Out[49]: [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 [50]: l[100]
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-50-3e7ce3111e95> in <module>()
----> 1 l[100]


IndexError: list index out of range
```

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

```
In [51]: new_list = ['a','e','x','b','c']

In [52]: #Show
         new_list

Out[52]: ['a', 'e', 'x', 'b', 'c']

In [53]: # Use reverse to reverse order (this is permanent!)
         new_list.reverse()

In [54]: new_list

Out[54]: ['c', 'b', 'x', 'e', 'a']

In [55]: # Use sort to sort the list (in this case alphabetical order, but for numb
         new_list.sort()

In [56]: new_list

Out[56]: ['a', 'b', 'c', 'e', 'x']
```

1.2 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 [57]: # 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 [60]: # Show
         matrix
```

```
Out[60]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Now 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 [61]: # Grab first item in matrix object
         matrix[0]

Out[61]: [1, 2, 3]

In [62]: # Grab first item of the first item in the matrix object
         matrix[0][0]

Out[62]: 1
```

2 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 [63]: # Build a list comprehension by deconstructing a for loop within a []
         first_col = [row[0] for row in matrix]

In [64]: first_col

Out[64]: [1, 4, 7]
```

We used 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 list section later on in this course!

```
In [ ]:
```

Dictionaries

August 20, 2017

1 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.

1.1 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 [13]: my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

In [4]: #Lets 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 [7]: #Can then even call methods on that value
        my_dict['key3'][0].upper()

Out[7]: 'ITEM0'
```

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

```
In [14]: my_dict['key1']

Out[14]: 123

In [15]: # Subtract 123 from the value
         my_dict['key1'] = my_dict['key1'] - 123

In [16]: #Check
         my_dict['key1']

Out[16]: 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 [17]: # Set the object equal to itself minus 123
         my_dict['key1'] -= 123
         my_dict['key1']

Out[17]: -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 [21]: # Create a new dictionary
         d = {}

In [22]: # Create a new key through assignment
         d['animal'] = 'Dog'

In [24]: # Can do this with any object
         d['answer'] = 42

In [25]: #Show
         d

Out[25]: {'animal': 'Dog', 'answer': 42}
```

1.2 Nesting with Dictionaries

Hopefully your 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 [26]: # Dictionary nested inside a dictionary nested in side a dictionary
         d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

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

```
In [29]: # Keep calling the keys
         d['key1']['nestkey']['subnestkey']

Out[29]: 'value'
```

1.3 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 [30]: # Create a typical dictionary
         d = {'key1':1,'key2':2,'key3':3}

In [35]: # Method to return a list of all keys
         d.keys()

Out[35]: ['key3', 'key2', 'key1']

In [36]: # Method to grab all values
         d.values()

Out[36]: [3, 2, 1]

In [33]: # Method to return tuples of all items  (we'll learn about tuples soon)
         d.items()

Out[33]: [('key3', 3), ('key2', 2), ('key1', 1)]
```

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

March 17, 2017

1 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.

1.1 Constructing Tuples

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

```
In [5]: # Can create a tuple with mixed types
        t = (1,2,3)

In [6]: # Check len just like a list
        len(t)

Out[6]: 3

In [8]: # Can also mix object types
        t = ('one',2)

        # Show
        t

Out[8]: ('one', 2)

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

Out[4]: 'one'
```

```
In [11]: # Slicing just like a list
         t[-1]
```

```
Out[11]: 2
```

1.2 Basic Tuple Methods

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

```
In [12]: # Use .index to enter a value and return the index
         t.index('one')
```

```
Out[12]: 0
```

```
In [13]: # Use .count to count the number of times a value appears
         t.count('one')
```

```
Out[13]: 1
```

1.3 Immutability

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

```
In [14]: t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-14-93def5f9b4bd> 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 [15]: t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-15-799b3447c4d9> in <module>()
----> 1 t.append('nope')


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

1.4 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 tuple become 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 Files!

Files

August 21, 2017

1 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!

1.1 iPython Writing a File

```
In [6]: %%writefile test.txt
        Hello, this is a quick test file

Overwriting test.txt
```

1.2 Python Opening a file

We can open a file with the open() function. The open function also takes in arguments (also called parameters). Lets see how this is used:

```
In [14]: # Open the text.txt we made earlier
         my_file = open('test.txt')

In [15]: # We can now read the file
         my_file.read()

Out[15]: 'Hello, this is a quick test file'

In [16]: # But what happens if we try to read it again?
         my_file.read()

Out[16]: ''
```

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

In [19]: # Now read again
         my_file.read()

Out[19]: 'Hello, this is a quick test file'
```

In order to not have to reset every time, we can also use 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 [26]: # Readlines returns a list of the lines in the file.
         my_file.readlines()

Out[26]: ['Hello, this is a quick test file']
```

1.3 Writing to a File

By default, using 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 [39]: # Add a second argument to the function, 'w' which stands for write
         my_file = open('test.txt','w+')

In [40]: # Write to the file
         my_file.write('This is a new line')

In [43]: # Read the file
         my_file.read()

Out[43]: 'This is a new line'
```

1.4 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 [44]: %%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 [45]: 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. Its 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 memor
3.) Notice the indent on the second line for print. This whitespace is required in
```

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

```
In [46]: # Pertaining to the first point above
         for asdf in open('test.txt'):
             print asdf
```

```
First Line

Second Line
```

```
In [ ]:
```

Sets and Booleans

March 17, 2017

1 Set and Booleans

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

1.1 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 [3]: # We add to sets with the add() method
        x.add(1)

In [4]: #Show
        x

Out[4]: {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 [5]: # Add a different element
        x.add(2)

In [6]: #Show
        x

Out[6]: {1, 2}

In [7]: # Try to add the same element
        x.add(1)

In [9]: #Show
        x

Out[9]: {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 [10]: # Create a list with repeats
         l = [1,1,2,2,3,4,5,6,1,1]

In [12]: # Cast as set to get unique values
         set(l)

Out[12]: {1, 2, 3, 4, 5, 6}
```

1.2 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 [13]: # Set object to be a boolean
         a = True

In [16]: #Show
         a

Out[16]: True
```

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

```
In [17]: # Output is boolean
         1 > 2

Out[17]: False
```

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

```
In [18]: # None placeholder
         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!

```
In [ ]:
```