01_python_basics

January 6, 2019

1 Python basics

Python is an interpreted, high-level, **general-purpose** programming language. It can be easy to pick up whether you're a first time programmer or you're experienced with other languages. Please find below some useful links.

https://www.python.org
https://docs.python.org/3.6/index.html
https://jupyter.org/

1.1 Files and folder path

```
In [ ]: pwd # current working folder path
```

```
In [1]: %%writefile my_file.txt
        Statisticians are the coolest.
```

```
Writing my_file.txt
```

```
In [2]: my_file = open(file='my_file.txt')
        print(my_file.read()) # read the file
        my_file.seek(0) # Seek to the start of file
        print(my_file.readlines()) # returns a list of the lines in the file
        my_file.close() # When you have finished using a file, it is always good practice to c
```

```
Statisticians are the coolest.
['Statisticians are the coolest.']
```

1.1.1 Caution !!

Opening a file with `'w'` or `'w+'` truncates the original, meaning that anything that was in the original file will be **deleted**. It allows us to read and write to the file though.

```
In [3]: my_file = open('my_file.txt','w+'); print(my_file.readlines())
        my_file.write('This is a new line')
        my_file.read()
        my_file.seek(0)
        print(my_file.readlines())
        my_file.close()
```

```
[]
['This is a new line']
```

1.1.2 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. We can also append with `%%writefile`.

```
In [4]: my_file = open('my_file.txt','a+')
        my_file.write('\nAdding a second line to the text file.')
        my_file.write('\nAdding a third line to the text file.')
        my_file.seek(0)
        print(my_file.readlines())
        my_file.close()
```

```
['This is a new line\n', 'Adding a second line to the text file.\n', 'Adding a third line to t
```

```
In [5]: %%writefile -a my_file.txt

        Adding a fourth line using writefile.
        Adding a fifth line using writefile.
```

```
Appending to my_file.txt
```

1.2 Types of objects

- **Integers (int):** whole numbers, e.g. `2  200  20000` etc.
- **Floating point (float):** numbers with a decimal point, e.g. `2.3  200.50  2000.56` etc.
- **Strings (str):** ordered sequence of characters, e.g. `"hello","Rakesh","2000"` etc.
- **Lists (list):** ordered sequence of objects, e.g. `[2,"hello",2.2]`
- **Dictionaries (dict):** unordered key-value pairs, e.g. `{"my_key":"key_value", "first_name":"Rakesh"}`
- **Tuples (tup):** ordered immutable sequence of objects, e.g. `(2,"hello",2.2)`
- **Sets (set):** unordered collection of unique objects, e.g. `("a","b")`

- **Booleans (bool):** logical value indicating `True`, or `False`

1.3 Variable assignments

We use a single = sign to assign values/labels to variables (`variable_name = object`). The names you use when creating these labels need to follow a few rules:

- Names can't start with a number.
- Spaces are not allowed in the name, use _ instead.
- Can't use any of the following symbols - `'",<>/?|\()!@#$%^&*~-+`.

- The best practice to name them is in `lowercase`.
- Avoid using the characters `l, O or I` as single character variable names.
- Avoid using words that have special meaning in Python like `list` and `str`.

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types and speed up developement time. Note that this may result in unexpected bugs, you need to be aware of. You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include

1.4 Math operations - (int & float)

Basically these are performed on integers and floating points.

```
In [ ]: print(2+1) # Addition
        print(2-1) # Subtraction
        print(2*2) # Multiplication
        print(3/2) # Division
        print(7%4) # Modulo,The % operator returns the remainder after division.
        print(7//4) # Floor Division,truncates the decimal without rounding and returns an int
        print(2**3) # Powers
        print(4**0.5) # Can also do roots this way
        print(3+10*10-3) # Order of Operations followed in Python
        print((2+8)*(13-3)) # Can use parentheses to specify orders
```

```
In [ ]: # assigning variables
        my_dogs = 2; print(my_dogs); print(type(my_dogs))
        my_dogs = ['Sammy', 'Frankie']; print(my_dogs); print(type(my_dogs))
```

```
In [ ]: # re-use assigned variables
        a = 5; print(a)
        a = a + a; print(a)
        a += 10; print(a)
        a *= 2; print(a)
```

```
In [ ]: # Simple Exercise
        my_income = 100
        tax_rate = 0.10
        my_tax_amount = my_income*tax_rate
        print(my_tax_amount); type(my_tax_amount)
```

1.5 String operations - (str)

Strings in Python are actually a ordered sequence. It keeps track of every element in the string as a sequence. *e.g.* 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, last letter* etc.).

```
In [ ]: print('hello') # Single word, we can also use double quote
        print('This is also a string') # Entire phrase
```

```
        print('Use \n to print a new line')
        print('Use \t to inserts a tab or space into a string')
        print('See what I mean?')
```

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

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

1.5.1 String basics, indexing and slicing

- `[]` - used after an object to call its **index**. Note that **indexing** starts at `0` for Python.
- `:` - used to perform **slicing** which grabs everything up to a designated point (`[start:stop:step]`)
- `len()` - used to check the **length** of a string.

```
In [ ]: my_string = 'Hello World' # assigning a string

        # Shows everything
        print(my_string)
        print(my_string[:])

        print(len(my_string)) # length of the string including spaces
        print(my_string[0]) # Shows first element
        print(my_string[1]) # Shows second element
        print(my_string[-1]) # Shows last letter (one index behind 0 so it loops back around)
        print(my_string[1:]) # Shows everything post the first index
        print(my_string[:3]) # Shows everything up to the 3rd index
        print(my_string[:-1]) # Shows everything but the last letter

        print(my_string[::1]) # Shows everything, but jumps in steps = 1
        print(my_string[::2]) # Shows everything, but jumps in steps = 2
        print(my_string[::-1]) # trick to print the string backwards
```

1.5.2 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.

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

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

```
In [ ]: my_string = my_string+' good morning !!' # Concatenate strings
        print(my_string)
        print('a'*10) #  multiplication symbol to create repetition
```

1.5.3 String methods

Objects in Python usually have built-in methods. These methods are functions inside the object that can perform actions or commands on the object itself i.e.

`object.method(parameters)`, where `parameters` are extra arguments we can pass into the method.

```
In [ ]: print(my_string.upper()) # Upper Case
        print(my_string.lower()) # Lower case
        print(my_string.split()) # Split by blank space (default)
        print(my_string.split('W')) # Split by a specific (won't be included) element
```

1.5.4 String formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. e.g. for a quick comparison refer the code below :

`student_name, student_score = 'Rakesh', 70`
`student_name+' scored '+str(student_score)+' points.'` - **concatenation**
`f'{student_name} scored {student_score} points.'` - **string formatting**
There are 3 ways to perform string formatting.

- **Formatting with placeholders** - Oldest method, involves using the modulo character %(referred as the *string formatting operator*) to inject strings into your print statements.
- **Formatting with the `.format()` method** - A better way to format objects into your strings for print statements. Following are the advantages:
  - Inserted objects can be called by index position.
  - Inserted objects can be assigned keywords.
  - Inserted objects can be reused, avoiding duplication
  - ...
  - Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more
  - You can pass an optional <,^, or > to set a left, center or right alignment
  - You can precede the aligment operator with a padding character
- **Formatted String Literals** (`f-strings`) - Newest method, introduced in Python 3.6, `f-strings` offer several benefits over the older `.format()` string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through `.format(var)`.

examples:

```
In [ ]: # Formatting with placeholders
        student_name, student_score = 'Rakesh', 70
        print("Name of the student is %s." % student_name) # single input
        print("%s has scored %s in the final exam." % (student_name, student_score)) # multipl
```

```
In [ ]: # Formatting with the .format() method
        print('Name of the student is {}'.format(student_name))
        print('{0} has scored {1} in the final exam.'.format(student_name,student_score))
```

```
In [ ]: # Formatting with f-strings
        print(f"Name of the student is {student_name}")
        print(f"{student_name} has scored {student_score} in the final exam.")
```

1.6 Lists operations - (list)

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. Lists are constructed with brackets `[]` and `commas` separating every element in the list. It supports **indexing** and **slicing**. Lists can be nested and also have a variety of useful methods that can be called off of them.

```
In [ ]: my_list = [1,2,3] # Assign a list to an variable named my_list
        my_list = ['A string',23,100.232,'o'] # lists can actually hold different object types
        print(my_list)
        print(len(my_list))
```

1.6.1 List basics, indexing & slicing

Indexing and slicing work just like in `strings`.

```
In [ ]: my_list = ['a','b','c','3','4']
        print(my_list[0]) # Grab element at index 0
        print(my_list[1:]) # Grab index 1 and everything past it
        print(my_list + ['new item']) # Use '+' to concatenate lists, just like we did for str
        print(my_list*2) # Make the list double
```

1.6.2 List methods

Lists in Python 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)
- they have no fixed type constraint (like we've seen above).

few basic methods are mentioned below:

- `append`: Adds an item to the end of a list permanently
- `pop`: Pops off an item from the list. By default pop takes off the last index, but we can also specify which index to pop off.
- `reverse`: Reverse order permanently.
- `sort`: Sorts the list (**letters** - alphabetical, **numbers**-ascending)

```
In [ ]: my_list.append('newly appended'); print(my_list)
        my_list.pop(); print(my_list)
        my_list.reverse(); print(my_list)
        my_list.sort(); print(my_list)
```

1.6.3 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.

```
In [ ]: list_1 = [1,2,3]
        list_2 = [4,5,6]
        list_3 = [7,8,9]
        nested_list = [list_1,list_2,list_3]
        print(nested_list)
        print(nested_list[0]) # Grab first item in matrix object
        print(nested_list[0][0]) # Grab first item of the first item in the matrix object
```

1.7 Dictionaries - (dict)

Dictionaries are unordered *mappings* for storing objects. *Mappings* are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. A Python dictionary consists of a key and then an associated value. That value can be almost any Python object. It allows users to quickly grab objects without needing to know an index location. It's important to note that dictionaries are very flexible in the data types they can hold. For example:

```
In [ ]: my_dict = {'key_1':'value_1','key_2':'value_2'} # Make a dictionary with {} and :
        print(my_dict['key_2']) # Call values by their key

        my_dict = {'k1':123,'k2':[1,2,3],'k3':['item_1','item_2','item_3']}
        print(my_dict['k3'])
        print(my_dict['k3'][0])
        print(my_dict['k3'][0].upper()) # Can then even call methods on that value
        print(my_dict['k1'] - 123) # Subtract 123 from the value
        my_dict['k1'] -= 123
        # print(my_dict['k1'])
```

1.7.1 Nesting Dictionaries

Like lists dictionary can be nested inside a dictionary:

```
In [ ]: nested_dict = {'key':{'nested_key':{'sub_nested_key':'value'}}}
        print(nested_dict['key']['nested_key']['sub_nested_key'])
```

1.7.2 Dictionary Methods

There are a few methods we can call on a dictionary.

```
In [ ]: print(my_dict.keys()) # return a list of all keys
        print(my_dict.values()) # grab all values
        print(my_dict.items()) # return tuples of all items
```

1.8 Tuples - (tup)

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. It uses parenthesis : `(1,2,3)` To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary

```
In [ ]: my_tuple = (1,2,'three')
        print(my_tuple)
        print(len(my_tuple))

In [ ]: print(my_tuple[0]) # indexing
        print(my_tuple[-1]) # Slicing
```

1.8.1 Tuple Methods

Tuples have built-in methods, but not as many as lists do.

```
In [ ]: my_tuple.index('three') # enter a value and return the index
        my_tuple.count(2) # count the number of times a value appears
```

1.9 Set and Booleans - (set & bool)

Sets are an unordered collection of **unique** elements. We can cast a `list` with multiple repeat elements to a `set` to get the unique elements. 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.

```
In [ ]: # sets
        my_set = set()
        my_set.add(1); print(my_set)
        my_set.add(2); print(my_set)
        my_set.add(1); print(my_set) # Try to add the same element
        print(set([1,1,2,2,3,4,5,6,1,1])) # takes unique elements only

In [ ]: # booleans
        my_bool = True; print(my_bool)
        print(type(my_bool))
        print(my_bool*1)
        print(1 > 2) # Output is boolean
```