This tutorial is accredited to **kenjyco** and available at https://gist.github.com/kenjyco (https://gist.github.com/kenjyco).

Right-click -> "save link as" [https://gist.githubusercontent.com/kenjyco/69eeb503125035f21a9d/raw/learning-python3.ipynb][learning-python3.ipynb] (https://gist.githubusercontent.com/kenjyco/69eeb503125035f21a9d/raw/learning-python3.ipynb%5D%5Blearning-python3.ipynb%5D) to get most up-to-date version of this notebook file.

Quick note about Jupyter cells

When you are editing a cell in Jupyter notebook, you need to re-run the cell by pressing **<Shift> + <Enter>**. This will allow changes you made to be available to other cells.

Python objects, basic types, and variables

Everything in Python is an **object** and every object in Python has a **type**. Some of the basic types include:

- **int** (integer; a whole number with no decimal place)
  - 10
  - -3
- **float** (float; a number that has a decimal place)
  - 7.41
  - -0.006
- **str** (string; a sequence of characters enclosed in single quotes, double quotes, or triple quotes)
  - 'this is a string using single quotes'
  - "this is a string using double quotes"
  - '''this is a triple quoted string using single quotes'''
  - """this is a triple quoted string using double quotes"""
- **bool** (boolean; a binary value that is either true or false)
  - True
  - False
- **NoneType** (a special type representing the absence of a value)
  - None

In [1]:
```
2 * 4 * 4
```
Out[1]:
```
32
```

Basic operators

In Python, there are different types of **operators** (special symbols) that operate on different values. Some of the basic operators include:

- arithmetic operators

    - `+` (addition)
    - `-` (subtraction)
    - `*` (multiplication)
    - `/` (division)
    - `**` (exponent)
- assignment operators
    - `=` (assign a value)
    - `+=` (add and re-assign; increment)
    - `-=` (subtract and re-assign; decrement)
    - `*=` (multiply and re-assign)
- comparison operators (return either `True` or `False`)
    - `==` (equal to)
    - `!=` (not equal to)
    - `<` (less than)
    - `<=` (less than or equal to)
    - `>` (greater than)
    - `>=` (greater than or equal to)

When multiple operators are used in a single expression, **operator precedence** determines which parts of the expression are evaluated in which order. Operators with higher precedence are evaluated first (like PEMDAS in math). Operators with the same precedence are evaluated from left to right.

- `()` parentheses, for grouping
- `**` exponent
- `*`, `/` multiplication and division
- `+`, `-` addition and subtraction
- `==`, `!=`, `<`, `<=`, `>`, `>=` comparisons

> See https://docs.python.org/3/reference/expressions.html#operator-precedence (https://docs.python.org/3/reference/expressions.html#operator-precedence)

In [0]:
```python
# Assigning some numbers to different variables
num1 = 10
num2 = -3
num3 = 7.41
num4 = -.6
num5 = 7
num6 = 3
num7 = 11.11
```

In [0]:
```python
# Addition
num1 + num2
```

Out[2]:
```
7
```

```
In [0]:
# Subtraction
num2 - num3
```

```
Out[3]: -10.41
```

```
In [0]:
# Multiplication
num3 * num4
```

```
Out[4]: -4.446
```

```
In [0]:
# Division
num4 / num5
```

```
Out[5]: -0.08571428571428572
```

```
In [0]:
# Exponent
num5 ** num6
```

```
Out[6]: 343
```

```
In [0]:
# Increment existing variable
num7 += 4
num7
```

```
Out[7]: 15.11
```

```
In [0]:
# Decrement existing variable
num6 -= 2
num6
```

```
Out[8]: 1
```

```
In [0]:
# Multiply & re-assign
num3 *= 5
num3
```

```
Out[9]: 37.05
```

```
In [0]:
# Assign the value of an expression to a variable
num8 = num1 + num2 * num3
num8
```

```
Out[10]: -101.14999999999999
```

```
In [0]:
# Are these two expressions equal to each other?
num1 + num2 == num5
```

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

```
In [0]:
# Are these two expressions not equal to each other?
num3 != num4
```

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

```
In [0]:
# Is the first expression less than the second expression?
num5 < num6
```

```
Out[13]: False
```

In [23]:
```python
# Is this expression True?
5 > 3 > 1
```
Out[23]:
```
True
```
In [0]:
```python
# Is this expression True?
5 > 3 < 4 == 3 + 1
```
Out[15]:
```
True
```
In [0]:
```python
# Assign some strings to different variables
simple_string1 = 'an example'
simple_string2 = "oranges "
```
In [0]:
```python
# Addition
simple_string1 + ' of using the + operator'
```
Out[17]:
```
'an example of using the + operator'
```
In [0]:
```python
# Notice that the string was not modified
simple_string1
```
Out[18]:
```
'an example'
```
In [0]:
```python
# Multiplication
simple_string2 * 4
```
Out[19]:
```
'oranges oranges oranges oranges '
```
In [0]:
```python
# This string wasn't modified either
simple_string2
```
Out[20]:
```
'oranges '
```
In [0]:
```python
# Are these two expressions equal to each other?
simple_string1 == simple_string2
```
Out[21]:
```
False
```
In [0]:
```python
# Are these two expressions equal to each other?
simple_string1 == 'an example'
```
Out[22]:
```
True
```
In [0]:
```python
# Add and re-assign
simple_string1 += ' that re-assigned the original string'
simple_string1
```
Out[23]:
```
'an example that re-assigned the original string'
```
In [0]:
```python
# Multiply and re-assign
simple_string2 *= 3
simple_string2
```
Out[24]:
```
'oranges oranges oranges '
```
In [0]:
```python
# Note: Subtraction, division, and decrement operators do not apply to s
```

Basic containers

> Note: **mutable** objects can be modified after creation and **immutable** objects cannot.

Containers are objects that can be used to group other objects together. The basic container types include:

- **str** (string: immutable; indexed by integers; items are stored in the order they were added)
- **list** (list: mutable; indexed by integers; items are stored in the order they were added)
  - [3, 5, 6, 3, 'dog', 'cat', False]
- **tuple** (tuple: immutable; indexed by integers; items are stored in the order they were added)
  - (3, 5, 6, 3, 'dog', 'cat', False)
- **set** (set: mutable; not indexed at all; items are NOT stored in the order they were added; can only contain immutable objects; does NOT contain duplicate objects)
  - {3, 5, 6, 3, 'dog', 'cat', False}
- **dict** (dictionary: mutable; key-value pairs are indexed by immutable keys; items are NOT stored in the order they were added)
  - {'name': 'Jane', 'age': 23, 'fav_foods': ['pizza', 'fruit', 'fish']}

When defining lists, tuples, or sets, use commas (,) to separate the individual items. When defining dicts, use a colon (:) to separate keys from values and commas (,) to separate the key-value pairs.

Strings, lists, and tuples are all **sequence types** that can use the `+`, `*`, `+=`, and `*=` operators.

In [0]:

```python
# Assign some containers to different variables
list1 = [3, 5, 6, 3, 'dog', 'cat', False]
tuple1 = (3, 5, 6, 3, 'dog', 'cat', False)
set1 = {3, 5, 6, 3, 'dog', 'cat', False}
dict1 = {'name': 'Jane', 'age': 23, 'fav_foods': ['pizza', 'fruit', 'fis
```

In [0]:

```python
# Items in the list object are stored in the order they were added
list1
```

Out[27]:

```
[3, 5, 6, 3, 'dog', 'cat', False]
```

In [0]:

```python
# Items in the tuple object are stored in the order they were added
tuple1
```

Out[28]:

```
(3, 5, 6, 3, 'dog', 'cat', False)
```

In [0]:

```python
# Items in the set object are not stored in the order they were added
# Also, notice that the value 3 only appears once in this set object
set1
```

Out[29]:

```
{3, 5, 6, False, 'cat', 'dog'}
```

In [0]:
```
# Items in the dict object are not stored in the order they were added
dict1
```
Out[30]:
```
{'name': 'Jane', 'age': 23, 'fav_foods': ['pizza', 'fruit', 'fish']}
```
In [0]:
```
# Add and re-assign
list1 += [5, 'grapes']
list1
```
Out[31]:
```
[3, 5, 6, 3, 'dog', 'cat', False, 5, 'grapes']
```
In [0]:
```
# Add and re-assign
tuple1 += (5, 'grapes')
tuple1
```
Out[32]:
```
(3, 5, 6, 3, 'dog', 'cat', False, 5, 'grapes')
```
In [0]:
```
[1, 2, 3] + [3, 4, 5]
```
Out[33]:
```
[1, 2, 3, 3, 4, 5]
```
In [0]:
```
# Multiply
[1, 2, 3, 4] * 2
```
Out[34]:
```
[1, 2, 3, 4, 1, 2, 3, 4]
```
In [0]:
```
# Multiply
(1, 2, 3, 4) * 3
```
Out[35]:
```
(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

Accessing data in containers

For strings, lists, tuples, and dicts, we can use **subscript notation** (square brackets) to access data at an index.

- strings, lists, and tuples are indexed by integers, **starting at 0** for first item
  - these sequence types also support accesing a range of items, known as **slicing**
  - use **negative indexing** to start at the back of the sequence
- dicts are indexed by their keys

> Note: sets are not indexed, so we cannot use subscript notation to access data elements.

In [0]:
```
list1
```
Out[36]:
```
[3, 5, 6, 3, 'dog', 'cat', False, 5, 'grapes']
```
In [0]:
```
# Access the first item in a sequence
list1[0]
```
Out[37]:
```
3
```

```
In [0]: tuple1
```

```
Out[80]: (3, 5, 6, 3, 'dog', 'cat', False, 5, 'grapes')
```

```
In [0]: # Access the last item in a sequence
        tuple1[-1]
```

```
Out[38]: 'grapes'
```

```
In [0]: simple_string1
```

```
Out[82]: 'an example that re-assigned the original string'
```

```
In [0]: # Access a range of items in a sequence
        simple_string1[3:8]
```

```
Out[39]: 'examp'
```

```
In [0]: # Access a range of items in a sequence
        tuple1[:-3]
```

```
Out[40]: (3, 5, 6, 3, 'dog', 'cat')
```

```
In [0]: # Access a range of items in a sequence
        list1[4:]
```

```
Out[41]: ['dog', 'cat', False, 5, 'grapes']
```

```
In [0]: dict1
```

```
Out[83]: {'name': 'Jane', 'age': 23, 'fav_foods': ['pizza', 'fruit', 'fish']}
```

```
In [0]: # Access an item in a dictionary
        dict1['name']
```

```
Out[42]: 'Jane'
```

```
In [0]: # Access an element of a sequence in a dictionary
        dict1['fav_foods'][2]
```

```
Out[43]: 'fish'
```

Python built-in functions and callables

A **function** is a Python object that you can "call" to **perform an action** or compute and **return another object**. You call a function by placing parentheses to the right of the function name. Some functions allow you to pass **arguments** inside the parentheses (separating multiple arguments with a comma). Internal to the function, these arguments are treated like variables.

Python has several useful built-in functions to help you work with different objects and/or your environment. Here is a small sample of them:

- **type(obj)** to determine the type of an object
- **len(container)** to determine how many items are in a container
- **sorted(container)** to return a new list from a container, with the items sorted

- **sum(container)** to compute the sum of a container of numbers
- **min(container)** to determine the smallest item in a container
- **max(container)** to determine the largest item in a container
- **abs(number)** to determine the absolute value of a number

> Complete list of built-in functions: https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html)

There are also different ways of defining your own functions and callable objects that we will explore later.

In [0]:
```
# Use the type() function to determine the type of an object
type(simple_string1)
```
Out[44]:
```
str
```

In [0]:
```
# Use the len() function to determine how many items are in a container
len(dict1)
```
Out[45]:
```
3
```

In [0]:
```
# Use the len() function to determine how many items are in a container
len(simple_string2)
```
Out[46]:
```
24
```

In [0]:
```
# Use the sorted() function to return a new list from a container, with
sorted([10, 1, 3.6, 7, 5, 2, -3])
```
Out[47]:
```
[-3, 1, 2, 3.6, 5, 7, 10]
```

In [0]:
```
# Use the sorted() function to return a new list from a container, with
# - notice that capitalized strings come first
sorted(['dogs', 'cats', 'zebras', 'Chicago', 'California', 'ants', 'mice
```
Out[48]:
```
['California', 'Chicago', 'ants', 'cats', 'dogs', 'mice', 'zebras']
```

In [0]:
```
# Use the sum() function to compute the sum of a container of numbers
sum([10, 1, 3.6, 7, 5, 2, -3])
```
Out[49]:
```
25.6
```

In [0]:
```
# Use the min() function to determine the smallest item in a container
min([10, 1, 3.6, 7, 5, 2, -3])
```
Out[50]:
```
-3
```

In [0]:
```
# Use the min() function to determine the smallest item in a container
min(['g', 'z', 'a', 'y'])
```
Out[51]:
```
'a'
```

In [0]:
```
# Use the max() function to determine the largest item in a container
max([10, 1, 3.6, 7, 5, 2, -3])
```
Out[52]:
```
10
```
In [0]:
```
# Use the max() function to determine the largest item in a container
max('gibberish')
```
Out[53]:
```
's'
```
In [0]:
```
# Use the abs() function to determine the absolute value of a number
abs(10)
```
Out[54]:
```
10
```
In [0]:
```
# Use the abs() function to determine the absolute value of a number
abs(-12)
```
Out[55]:
```
12
```

Python object attributes (methods and properties)

Different types of objects in Python have different **attributes** that can be referred to by name (similar to a variable). To access an attribute of an object, use a dot (.) after the object, then specify the attribute (i.e. obj.attribute)

When an attribute of an object is a callable, that attribute is called a **method**. It is the same as a function, only this function is bound to a particular object.

When an attribute of an object is not a callable, that attribute is called a **property**. It is just a piece of data about the object, that is itself another object.

The built-in dir() function can be used to return a list of an object's attributes.

Some methods on list objects

- **.append(item)** to add a single item to the list
- **.extend([item1, item2, ...])** to add multiple items to the list
- **.remove(item)** to remove a single item from the list
- **.pop()** to remove and return the item at the end of the list
- **.pop(index)** to remove and return an item at an index

In [0]:
```

```

Some methods on dict objects

- **.update(dict2)** to add all keys and values from another dict to the dict
- **.keys()** to return a list of keys in the dict
- **.values()** to return a list of values in the dict
- **.items()** to return a list of key-value pairs (tuples) in the dict

In [0]:

```python

```

Control Flow Statements

The key thing to note about Python's control flow statements and program structure is that it uses *indentation* to mark blocks. Hence the amount of white space (space or tab characters) at the start of a line is very important. This generally helps to make code more readable but can catch out new users of python.

Conditionals

If

```
if some_condition:
    code block
```

In [0]:

```python
x = 12
if x > 10:
    print("Hello")
```

```
Hello
```

If-else

```
if some_condition:
    algorithm
else:
    algorithm
```

In [0]:

```python
x = 12
if 10 < x < 11:
    print("hello")
else:
    print("world")
```

```
world
```

Else if

```
if some_condition:
    algorithm
elif some_condition:
    algorithm
else:
    algorithm
```

In [0]:
```python
x = 10
y = 12
if x > y:
    print("x>y")
elif x < y:
    print("x<y")
else:
    print("x=y")
```

```
x<y
```

if statement inside a if statement or if-elif or if-else are called as nested if statements.

In [0]:
```python
x = 10
y = 12
if x > y:
    print( "x>y")
elif x < y:
    print( "x<y")
    if x == 10:
        print ("x=10")
    else:
        print ("invalid")
else:
    print ("x=y")
```

```
x<y
x=10
```

Loops

For

```
for variable in something:
    algorithm
```

When looping over integers the **range()** function is useful which generates a range of integers:

- range(n) = 0, 1, ..., n-1
- range(m,n)= m, m+1, ..., n-1
- range(m,n,s)= m, m+s, m+2s, ..., m + ((n-m-1)//s) * s

In [0]:
```
for ch in 'abc':
    print(ch)
total = 0
for i in range(5):
    total += i
for i,j in [(1,2),(3,1)]:
    total += i**j
print("total =",total)
```

```
a
b
c
total = 14
```

In the above example, i iterates over the 0,1,2,3,4. Every time it takes each value and executes the algorithm inside the loop. It is also possible to iterate over a nested list illustrated below.

In [0]:
```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list1 in list_of_lists:
        print(list1)
```

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

A use case of a nested for loop in this case would be,

In [0]:
```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
total=0
for list1 in list_of_lists:
    for x in list1:
        total = total+x
print(total)
```

```
45
```

While

```
while some_condition:
    algorithm
```

In [0]:
```
i = 1
while i < 3:
    print(i ** 2)
    i = i+1
print('Bye')
```

```
1
4
Bye
```

Break

As the name says. It is used to break out of a loop when a condition becomes true when executing the loop.

In [0]:
```python
for i in range(100):
    print(i)
    if i >= 7:
        break
```

```
0
1
2
3
4
5
6
7
```

Continue

This continues the rest of the loop. Sometimes when a condition is satisfied there are chances of the loop getting terminated. This can be avoided using continue statement.

In [0]:
```python
for i in range(10):
    if i > 4:
        print("Ignored",i)
        continue
    # this statement is not reach if i > 4
    print("Processed",i)
```

```
Processed 0
Processed 1
Processed 2
Processed 3
Processed 4
Ignored 5
Ignored 6
Ignored 7
Ignored 8
Ignored 9
```

Functions

Functions can represent mathematical functions. More importantly, in programmming functions are a mechansim to allow code to be re-used so that complex programs can be built up out of simpler parts.

This is the basic syntax of a function

```python
def funcname(arg1, arg2,... argN):
    ''' Document String'''
    statements
    return <value>
```

Read the above syntax as, A function by name "funcname" is defined, which accepts arguements "arg1,arg2,....argN". The function is documented and it is '''Document String'''. The function after executing the statements returns a "value".

Return values are optional (by default every function returns **None** if no return statement is executed)

```
In [0]: print("Hello Jack.")
        print("Jack, how are you?")
```

```
Hello Jack.
Jack, how are you?
```

Instead of writing the above two statements every single time it can be replaced by defining a function which would do the job in just one line.

Defining a function firstfunc().

```
In [0]: def firstfunc():
            print("Hello Jack.")
            print("Jack, how are you?")
        firstfunc() # execute the function
```

```
Hello Jack.
Jack, how are you?
```

firstfunc() every time just prints the message to a single person. We can make our function **firstfunc()** to accept arguements which will store the name and then prints respective to that accepted name. To do so, add a argument within the function as shown.

```
In [0]: def firstfunc(username):
            print("Hello %s." % username)
            print(username + ',' ,"how are you?")
```

```
In [0]: name1 = 'sally' # or use input('Please enter your name : ')
```

So we pass this variable to the function **firstfunc()** as the variable username because that is the variable that is defined for this function. i.e name1 is passed as username.

```
In [0]: firstfunc(name1)
```

```
Hello sally.
sally, how are you?
```

Return Statement

When the function results in some value and that value has to be stored in a variable or needs to be sent back or returned for further operation to the main algorithm, a return statement is used.

```
In [0]: def times(x,y):
            z = x*y
            return z
```

The above defined **times( )** function accepts two arguements and return the variable z which contains the result of the product of the two arguements

In [0]:
```
c = times(4,5)
print(c)
```
```
20
```

The z value is stored in variable c and can be used for further operations.

Instead of declaring another variable the entire statement itself can be used in the return statement as shown.

In [0]:
```
def times(x, y):
    '''This multiplies the two input arguments'''
    return x*y
```

In [0]:
```
c = times(4, 5)
print(c)
```
```
20
```

Since the **times( )** is now defined, we can document it as shown above. This document is returned whenever **times( )** function is called under **help( )** function.

In [0]:
```
help(times)
```
```
Help on function times in module __main__:

times(x, y)
    This multiplies the two input arguments
```

Multiple variable can also be returned as a tuple. However this tends not to be very readable when returning many value, and can easily introduce errors when the order of return values is interpreted incorrectly.

In [0]:
```
eglist = [10, 50, 30, 12, 6, 8, 100]
```

In [0]:
```
def egfunc(eglist):
    highest = max(eglist)
    lowest = min(eglist)
    first = eglist[0]
    last = eglist[-1]
    return highest, lowest, first, last
```

If the function is just called without any variable for it to be assigned to, the result is returned inside a tuple. But if the variables are mentioned then the result is assigned to the variable in a particular order which is declared in the return statement.

In [0]:
```
egfunc(eglist)
```
Out[78]:
```
(100, 6, 10, 100)
```

In [0]:
```
a, b, c, d = egfunc(eglist)
print(' a =', a, ' b =', b, ' c =', c, ' d =', d)
```

```
 a = 100  b = 6  c = 10  d = 100
```

Positional arguments and keyword arguments to callables

You can call a function/method in a number of different ways:

- `func()` : Call `func` with no arguments
- `func(arg)` : Call `func` with one positional argument
- `func(arg1, arg2)` : Call `func` with two positional arguments
- `func(arg1, arg2, ..., argn)` : Call `func` with many positional arguments
- `func(kwarg=value)` : Call `func` with one keyword argument
- `func(kwarg1=value1, kwarg2=value2)` : Call `func` with two keyword arguments
- `func(kwarg1=value1, kwarg2=value2, ..., kwargn=valuen)` : Call `func` with many keyword arguments
- `func(arg1, arg2, kwarg1=value1, kwarg2=value2)` : Call `func` with positonal arguments and keyword arguments

When using **positional arguments**, you must provide them in the order that the function defined them (the function's **signature**).

When using **keyword arguments**, you can provide the arguments you want, in any order you want, as long as you specify each argument's name.

When using positional and keyword arguments, positional arguments must come first.

In [0]:
```

```

References

- https://try.jupyter.org (https://try.jupyter.org)
- https://docs.python.org/3/tutorial/index.html (https://docs.python.org/3/tutorial/index.html)
- https://docs.python.org/3/tutorial/introduction.html (https://docs.python.org/3/tutorial/introduction.html)
- https://daringfireball.net/projects/markdown/syntax (https://daringfireball.net/projects/markdown/syntax)