GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Data Literacy

Tutor : Ghassan Al-Falouji
Supervisor : Prof. Dr. Roland Mandl

AUFSTIEG DURCH BILDUNG
OFFENE HOCHSCHULEN

In [1]:
```
## CSS coloring for the dataframe tables

from IPython.core.display import HTML
#
css = open('../style/style-table.css').read() + open('../style/style-notebook.css').read()
HTML('<style>{}</style>'.format(css))
```

Out[1]:

Python Script programming and iPython

Table of contents

What is Python ? [[1](#ref1)]

"*The best things in life are free!*"

Python is an open source, high-level, object oriented, interpreted, and general purpose dynamic programming language. It was create by Guido van Rossum during 1985-1990. Python interpreter source code is availbale inder the

The design philosophy of Python [[1]](#ref1) is:

- **Be consistent**, beautiful is better than ugly
- **Use existing libraries**, Complex is better than complicated.
- **Keep it simple and stupid (`KISS`)**, Simple is better than complex.
- **Avoid nested ifs**, Flat is better than nested.
- **Be clear**, Explicit is better than implicit.
- **Seperate code into modules**, Sparse is better than dense.
- **Indenting for easy readability**, Readability counts.
- **Everything is an object**, Special cases aren't special enough to break the rules.

- **Good exception handler**, Errors should never pass silently.
- **If reuired, break the rules**, Although practicality beats purity.
- **Error logging and traceability**, Unless explicitly silenced.
- **Python syntax is simpler; however, many times we might take a longer timer to decipher it**, In the face of ambiguity, refuse the temptation to guess.
- **There is not only one way of schieving something**, Although that way may not be obvious at first unless you're Dutch.
- **Use existing libraries, again !**, There should be one-- and preferably only one --obvious way to do it.
- **If you can't explain in simple terms then you don't understand it well enought**, If the implementation is hard to explain, it's a bad idea.
- **There are auick/dirty ways to get the job done rather than trying too much to optimize**, Now is better than never.
- **Although there is a quick/dirty way, don't head in the path that will now allow a graceful way back**, Although never is often better than *right* now.
- **Be specific**, Namespaces are one honking great idea -- let's do more of those!
- **Simplicity**, If the implementation is easy to explain, it may be a good idea.

Python Features [[1](#ref1)]

Python is a interpreted, interactive, object-oriented, and high-level programming language. Python enjoys many features, such as:

1. **A simple language which is easy to learn**
   Python has a simple and elegant syntax, which is easy to read and implement.
2. **Free and open-source**
   It can be used freely and distributed even for commercial use.
3. **Portability**
   The code is cross platform, weather it was implemented for Windows, linux or Max OS.
4. **Extensible and Embedded**
   Python code is embeddable in other code in C/C++/Java. It's possible to combine pieces of C/C++ or other languages with Python code. This will give application high performance as well as scripting capabilities.
5. **A high-level, interpreted language**
   Unlike C/C++, programmers need not to worry about memory management and garbage collection, when Python is adopted as the programming language. At run time, it automatically converts source code to the language of computer.
6. **Lange standard libraries**
   Python has a number of standard libraries which makes it easy to program. Before you start implementing your own algorithm, look around ! you will find something that fits your requirements or at least a start point that you can tune for your needs.
7. **Object-oriented**
   Everything in Python is an object !

Installation [[1](#ref1)]

Installation information can be found in THIS LINK.

Another possibility is to install Anaconda, which will install for you Python, IPython, Jupyter and many basic packages.

Installing or Updating Packages

Using native Python is usually not enough/or too complex to complete your desired tasks, usually these is packages for almost every goal you may need Python for. These packages can/ has to be installed/updated explicitly. In `Anaconda` this can be performed either using the GUI, or by running the following bash code in your command-line/terminal:

- For installing

```
conda install package_name
```

- For updating

```
conda update package_name
```

But if you installed native Python (not Anaconda suit), you can install the desired package by running the following bash code:

- For installing and upgrading

```
pip install -U package_name
```

Basic Packages to Install

- **IPython** : is a command shell for interactive computing in multiple programming languages, originally developed for the Python.

In [ ]:
```
! pip install -U IPython
```

> This Jupyter notebook is running IPython. IPython gives a lot of flexibility to Native Python, such as running shell in Jupyter. This can be done, by preceding your shell code with `!`. Therefore if you want to install the package using the command-line/terminal, copy the previous line without the `!`.

- **NumPy** : Library for matrix/array manipulation

In [ ]:
```
! pip install -U numpy
```

- **Pandas** : Ease the import of datasets into tabulated format.

In [ ]:
```
! pip install -U pandas
```

- **Jupyter** : Your are now looking at jupyter notebooks !

In [ ]:
```
! pip install -U jupyter
```

- **Matplotlib** and **Seaborn** : Both libraries are necessary for visualization.

In [ ]:
```
! pip install -U matplotlib seaborn
```

Integrated Development Environments (IDEs) and Text Editors [[2](#ref2)]

IPython and its Jupyter notebook are example of a Python IDE. It is a great tool for code development stage, where you can document and test every piece of your code. Still you can develop your Python code on any text editor and run it in your command-line/terminal. As example: In the same directory of this notebook there is a `hello.py` Python script. Check it out, before you run it in your command line using :

In [6]:
```
! python hello.py
```

```
Calling main function
Hello World !
```

However, when building a big project/application, some developers prefer to using more richly featured IDE (at least with good *intellisence*). Here are some IDEs:

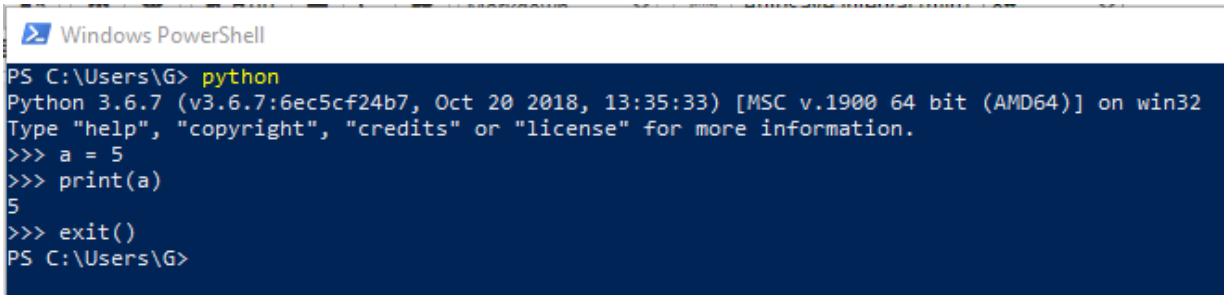
- **PyDev** (free), IDE built on Eclipse
- **PyCharm** (free for open source developers), IDE from JetBrain.
- **Python Tools** for Visual Studio (The addon is free, but VS is free for community version otherwise)
- **Visual Studio Code** (free), text editor with good intellisence for Windows users.
- **Spyder** (free), an IDE currently shipped with Anaconda, but can be installed also as stand-alone.
- **Komodo** IDE (commercial)
- ...

Community [[2](#ref2)]

- **Pydata** : A Google Group list for questions related to Python for data analysis and pandas
- **pystatsmodels** : For `statsmodels` or pandas-related questions
- Mailing list for **scikit-learn** (scikit-learn@python.org) and Machine learning in Python for general.
- **numpy-discussion** : For NumPy-related questions.
- **scipy-user** : For general SciPy or scientific Python questions.

Python Interpreter

The Python interpreter runs a program by executing one statement at a time. The standard interactive Python interpreter can be invoked on the command line with the python command:



- Help !

Variables, statements and expressions

Comment

- One line comment

In [1]:
```
# This is a comment
```

- Multiple line comment
  This method is used to write a manual/help of a function, which can be viewed when calling `help(my_func)` in Python (or `my_func?` in IPython).

In [2]:
```
"""
Here is the description of my function
"""
```

Out[2]:
```
'\nHere is the description of my function\n'
```

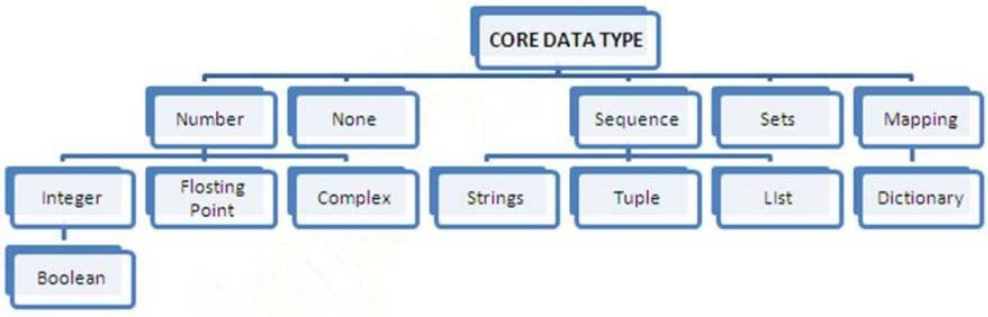
Variables

Variables are examples of *Identifiers*. *Identifiers* are names given to identify *something* (Variable, function, class, object, ...).

Rules for naming identifiers are:

- First character mush be an alphabet or an underscore.
- The rest of the identifier name can consist of letters, underscores, or digits.
- Identifier names are case-sensitive. As example `A` is not `a`.

source: *https://bit.ly/2lsgTYK*



Numeric

- Integer

In [3]:
```
# Define an integer variable
a = 5

# Print variable value
print(a)

# Print the type of the variable (a)
print(type(a))
```

```
5
<class 'int'>
```

- Float

In [4]:
```
# Define a float variable
b= 5.0

print("b = ",b)
print("\t has type: ",type(b))
```

```
b =  5.0
         has type:  <class 'float'>
```

- String

In [5]:
```
# Define a string variable
c = '5' # or "5"
print("c = ", c)
print("\t has type: ",type(c))
```

```
c =  5
         has type:  <class 'str'>
```

- Bool

In [6]:
```
# Define a string variable
d = True # or False
print("d = ", d)
print("\t has type: ",type(d))
```

```
d =  True
         has type:  <class 'bool'>
```

- Complex

In [9]:
```
# Define a complex variable
e = 1+5j # or complex(1,5)
print("e = ", e)
print("\t has type: ",type(e))
```

```
e =  (1+5j)
         has type:  <class 'complex'>
```

Converting between numeric data types

In [11]:
```
a = 5 # int
a = float(a) # convert to float

print(type(a))
```

```
<class 'float'>
```

In [14]:
```
b = "5.6"
```

```
b_1 = float(b)
b_2 = int(b_1)

print("b_1 = {0}\nb_2 = {1}".format(b_1, b_2))
```

```
b_1 = 5.6
b_2 = 5
```

Strings

In [26]:
```
str = 'Hello World!'

# ---- CODE HERE ----- #
```

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

> Notice that Python index starts from 0

Lists

A list contains items separated by commas and enclosed within square brackets (`[ ]`). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type, also it's possible to change/update list elements (called `mutable`).

In [27]:
```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

# ---- CODE HERE ----- #
```

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas and enclosed in (( )). Tuples can hold different typed items.

In [28]:
```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

# ---- CODE HERE ----- #
```

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple (tuples are **immutable**), which is not allowed. Similar case is possible with lists:

In [38]:
```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]

try:
    tuple[2] = 1000    # Invalid syntax with tuple
except TypeError as e:
    print("Error in executing:\n\t tuple[2]")
    print(e)
    pass

#
list[2] = 1000     # Valid syntax with list

#
print()
print(tuple)
print(list)
```

```
Error in executing:
        tuple[2]
'tuple' object does not support item assignment

('abcd', 786, 2.23, 'john', 70.2)
['abcd', 786, 1000, 'john', 70.2]
```

Dictionaries

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([ ]). For example:

In [42]:
```
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
```

```
print(tinydict)
print(tinydict.keys())
print(tinydict.values())

# Get the value of key 'code'
tinydict['code']
```

```
{'name': 'john', 'code': 6734, 'dept': 'sales'}
dict_keys(['name', 'code', 'dept'])
dict_values(['john', 6734, 'sales'])
```

Out[42]:
```
6734
```

Dictionaries have no concept of order among elements.

In [43]:
```
anoth_dict = {}
anoth_dict['one']=1 # key:'one', value:1
anoth_dict[5]='five'# key:5, value:'five'

print(anoth_dict)
```

```
{'one': 1, 5: 'five'}
```

In [44]:
```
# get the value of key 5
anoth_dict[5]
```

Out[44]:
```
'five'
```

Expressions

Simply ...

In [15]:
```
1+1
```

Out[15]:
```
2
```

In [16]:
```
'Hi'
```

Out[16]:
```
'Hi'
```

In [17]:
```
17*10
```

Out[17]:
```
170
```

In [18]:
```
3*'a'
```

Out[18]:
```
'aaa'
```

> COOL ! or ?

In [19]:
```
'a'+'b'
```

Out[19]:
```
'ab'
```

In [21]:
```
a = 10
a*a
```

Out[21]:
```
100
```

Input

Explore `input_ex.py` python code which you will find in the current directory:

In [ ]:
```
! pwd
```

- To run the python script in Jupyter (**Thanks to IPython**):

In [24]:
```
%run input_ex.py
```

```
Enter your name:        Ghassan Al-Falouji

Hello Ghassan Al-Falouji
```

- To see the code of a python script in Jupyter (**Thanks to IPython**):

In [ ]:
```
%load input_ex.py
```

Mathematical Operations

Arithmetic Operator

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2

% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

source: *https://bit.ly/2nhND43*

In [45]:
```
a = 21
b = 10
c = 0

c = a + b
print ("Line 1 - Value of c is ", c)

c = a - b
print ("Line 2 - Value of c is ", c)

c = a * b
print ("Line 3 - Value of c is ", c)

c = a / b
print ("Line 4 - Value of c is ", c)

c = a % b
print ("Line 5 - Value of c is ", c)
```

```
Line 1 - Value of c is  31
Line 2 - Value of c is  11
Line 3 - Value of c is  210
Line 4 - Value of c is  2.1
Line 5 - Value of c is  1
```

In [47]:
```
a = 2
b = 3
c = a**b
print ("Line 6 - Value of c is ", c)

a = 10
b = 4
c = a//b
print ("Line 7 - Value of c is ", c)
```

```
Line 6 - Value of c is  8
Line 7 - Value of c is  2
```

Comparison Operators

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

source: *https://bit.ly/2nhND43*

In [50]:
```
a = 21
b = 10
c = 0

if ( a == b ):
    print ("Line 1 - a is equal to b")
else:
    print ("Line 1 - a is not equal to b")
    pass

if ( a != b ):
    print ("Line 2 - a is not equal to b")
else:
    print ("Line 2 - a is equal to b")
    pass

if ( a < b ):
    print ("Line 3 - a is less than b" )
else:
    print ("Line 3 - a is not less than b")
    pass

if ( a > b ):
    print ("Line 4 - a is greater than b")
else:
    print ("Line 4 - a is not greater than b")
    pass
```

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
```

```
Line 3 - a is not less than b
Line 4 - a is greater than b
```

In [51]:
```
a = 5;
b = 20;
if ( a <= b ):
    print ("Line 6 - a is either less than or equal to  b")
else:
    print ("Line 6 - a is neither less than nor equal to  b")

if ( b >= a ):
    print ("Line 7 - b is either greater than  or equal to b")
else:
    print ("Line 7 - b is neither greater than  nor equal to b")
```

```
Line 6 - a is either less than or equal to  b
Line 7 - b is either greater than  or equal to b
```

Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

source: *https://bit.ly/2nhND43*

In [52]:
```
a = 21
b = 10
c = 0

c = a + b
print ("Line 1 - Value of c is ", c)

c += a
print ("Line 2 - Value of c is ", c)

c *= a
print ("Line 3 - Value of c is ", c )

c /= a
print ("Line 4 - Value of c is ", c )

c  = 2
c %= a
print ("Line 5 - Value of c is ", c)

c **= a
print ("Line 6 - Value of c is ", c)

c //= a
print ("Line 7 - Value of c is ", c)
```

```
Line 1 - Value of c is  31
Line 2 - Value of c is  52
Line 3 - Value of c is  1092
Line 4 - Value of c is  52.0
Line 5 - Value of c is  2
Line 6 - Value of c is  2097152
Line 7 - Value of c is  99864
```

Bitwise Operators

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<<	The left operands value is moved left by the	

Binary Left Shift	number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

source: https://bit.ly/2nhND43

In [53]:
```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

print("a = {}".format(bin(a)))
print("b = {}".format(bin(b)))

c = a & b;        # 12 = 0000 1100
print ("Line 1 - Value of c is {} = {}".format(c, bin(c)))

c = a | b;        # 61 = 0011 1101
print ("Line 2 - Value of c is {} = {}".format(c, bin(c)))

c = a ^ b;        # 49 = 0011 0001
print ("Line 3 - Value of c is {} = {}".format(c, bin(c)))

c = ~a;           # -61 = 1100 0011
print ("Line 4 - Value of c is {} = {}".format(c, bin(c)))

c = a << 2;       # 240 = 1111 0000
print ("Line 5 - Value of c is {} = {}".format(c, bin(c)))

c = a >> 2;       # 15 = 0000 1111
print ("Line 6 - Value of c is {} = {}".format(c, bin(c)))
```

```
a = 0b111100
b = 0b1101
Line 1 - Value of c is 12 = 0b1100
Line 2 - Value of c is 61 = 0b111101
Line 3 - Value of c is 49 = 0b110001
Line 4 - Value of c is -61 = -0b111101
Line 5 - Value of c is 240 = 0b11110000
Line 6 - Value of c is 15 = 0b1111
```

Logical Operators

Operator Description Example and Logical AND If both the operands are true then condition becomes true. (a and b) is true. or Logical OR If any of the two operands are non-zero then condition becomes true. (a or b) is true. not Logical NOT Used to reverse the logical state of its operand. Not(a and b) is false.

source: https://bit.ly/2nhND43

Membership

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

source: *https://bit.ly/2nhND43*

In [55]:
```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
    print ("Line 1 - a is available in the given list")
else:
    print ("Line 1 - a is not available in the given list")

if ( b not in list ):
    print ("Line 2 - b is not available in the given list")
else:
    print ("Line 2 - b is available in the given list")

a = 2
if ( a in list ):
    print ("Line 3 - a is available in the given list")
else:
    print ("Line 3 - a is not available in the given list")
```

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names

	for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

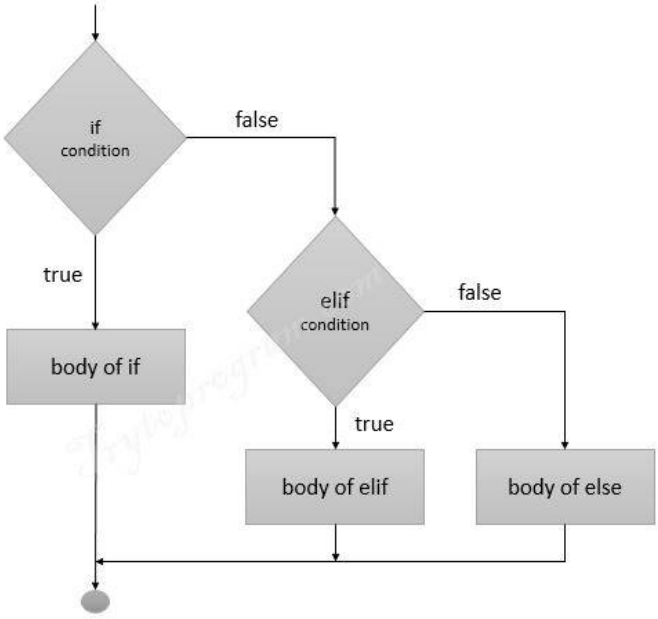
source: *https://bit.ly/2nhND43*

Conditional- and loop statements

`if` statement

A conditional statement that can be represented in the following flowchart:

source: https://bit.ly/2lu8p3a



- Example:

In [58]:
```
# ---- CODE HERE ----- #
```

```
Bye !
```

In [59]:
```
# ---- CODE HERE ----- #
```

```
Value expression is 100
Bye !
```

Nested **if .. else** statement

- Example:

In [81]:
```
import numbers
import six
```

In [82]:
```
# ---- CODE HERE ----- #
```

```
Variable is a string
exit
```

In [65]:
```
# To check number class methods
numbers.__dir__()
```

Out[65]:
```
['__name__',
 '__doc__',
 '__package__',
 '__loader__',
 '__spec__',
 '__file__',
 '__cached__',
 '__builtins__',
 'ABCMeta',
 'abstractmethod',
```

```
'__all__',
'Number',
'Complex',
'Real',
'Rational',
'Integral']
```

`for` statement

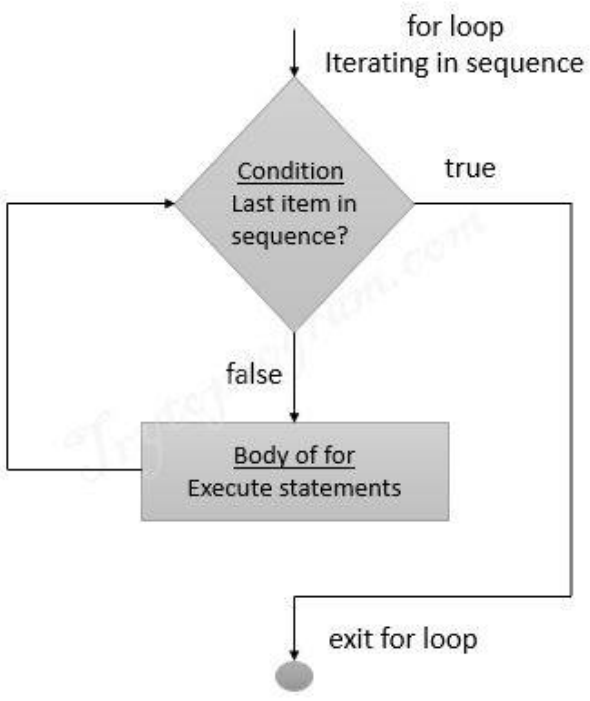
Python `for` loop is used for repeated execution of a group of statements for the desired number of times. It iterates over the items of lists, tuples, strings, the dictionaries and other iterable objects.

- Syntax:

```
for var in sequence:
  statement(s)
```

For loop can be presented in the following flow chart:

source: *https://bit.ly/2kopnzG*



Looping over list/tuple elements

In [85]:
```
my_list = ['Apple','Mango','Guava','Pineapple'] #('Apple','Mango','Guava','Pineapple')
i = 0

# ---- CODE HERE ----- #
```

```
element (0) : Apple
element (1) : Mango
element (2) : Guava
element (3) : Pineapple
```

The same previous code can be done with sparing the `i`:

In [84]:
```
my_list = ['Apple','Mango','Guava','Pineapple'] #('Apple','Mango','Guava
','Pineapple')

# ---- CODE HERE ----- #
```

```
element (0) : Apple
element (1) : Mango
element (2) : Guava
element (3) : Pineapple
```

Iterating over string characters

In [86]:
```
my_string = "HELLO"

# ---- CODE HERE ----- #
```

```
H
E
L
L
O
```

Iterating over a dictionary items

In [88]:
```
bikes = {"key_1":1, "key_2":2, "key_3":3, "key_4":4}

# ---- CODE HERE ----- #
```

```
key : key_1             value : 1
key : key_2             value : 2
key : key_3             value : 3
key : key_4             value : 4
```

Iterating over the keys

In [89]:
```
bikes = {"key_1":1, "key_2":2, "key_3":3, "key_4":4}

# ---- CODE HERE ----- #
```

```
key : key_1             value : 1
key : key_2             value : 2
key : key_3             value : 3
key : key_4             value : 4
```

Iterating over the values

In [91]:
```
bikes = {"key_1":1, "key_2":2, "key_3":3, "key_4":4}
```

```
# ---- CODE HERE ----- #
```

```
value : 1
value : 2
value : 3
value : 4
```

`range()` function

The `range( )` is a built-in Python function used for iterating over a sequence of numbers.

Syntax

- `range(n)` : will generate numbers from `0` to `(n-1)`
- `range(x,y)` : will generate numbers from `x` to `(y-1)`
- `range(start,end,step_size)` : will generate numbers from `start` to `end` with `step_size` as incremental factor in each iteration. `step_size` is default 1 if not explicitly mentioned.

In [1]:
```
list(range(9)) # Not working ? Because previously we have used list as a variable name, but it is
               # a reserved word in python. This is an example of bad naming.
               # What to do ? Restart the kernel ..  change the previous list variable to another
               # name, then rerun this code part.
```

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

In [3]:
```
list(range(2,5))
```

Out[3]:
```
[2, 3, 4]
```

In [4]:
```
list(range(1,10,2))
```

Out[4]:
```
[1, 3, 5, 7, 9]
```

- Example: Looping over a list using the `range()`

In [6]:
```
my_list = ['Soccer','Cricket','Golf']

print("length of my_list = ", len(my_list))

print()

# loop
for i in range(len(my_list)):
    print("{0} : {1}".format(i, my_list[i]))
    pass
```

```
length of my_list =  3

0 : Soccer
1 : Cricket
```

```
2 : Golf
```

Exercise

Below, given a two dimensional array `a`.

Find the sum of each row.

In [11]:
```
a  = [[1, 2, 3, 4, 5],
      [6, 7, 8, 9, 10]]
a
```

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

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

`while` statement

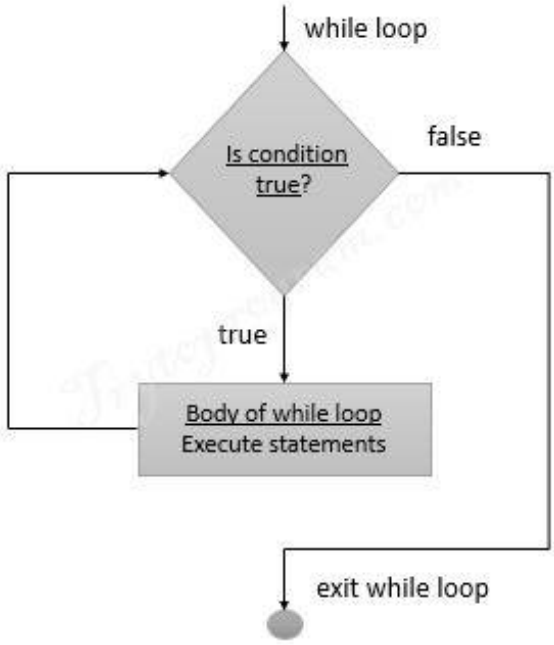
Loops are either infinite or conditional. Python `while` loop keeps reiterating a block of code defined inside it until the desired condition is met.

The `while` loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true.

- Syntax:

```
while (expression):
  statement(s)
```

source: https://bit.ly/2kopnzG



- Example:

In [7]:
```
i = 5
# ---- CODE HERE ----- #
```

```
5 : This is while loop
6 : This is while loop
7 : This is while loop
```

Exercise

Using `while` loop extract all even numerics from the `in_list`.

In [1]:
```
import numpy as np # Module used to generate random int
```

In [2]:
```
in_list = list(np.random.randint(0,21,15))
in_list
```

Out[2]:
```
[14, 15, 7, 10, 20, 14, 19, 19, 10, 17, 2, 6, 1, 13, 10]
```

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

`break` and `continue` statements

In Python, break and continue statements can alter the flow of a normal loop.

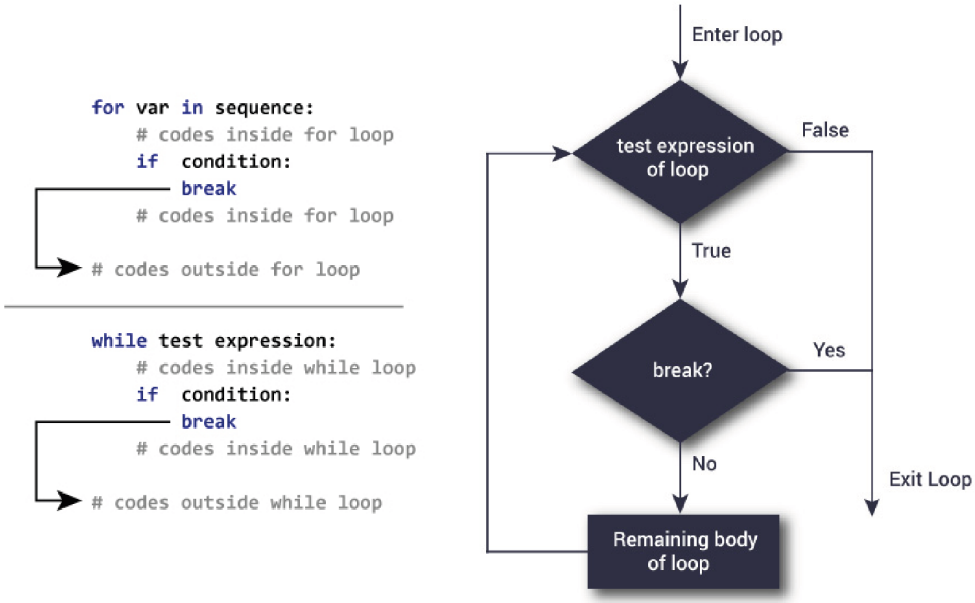
Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

`break` statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

source: *https://bit.ly/2XE6MgD*



- Example:

In [5]:
```
for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

```
s
t
```

```
r
The end
```

`continue` statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

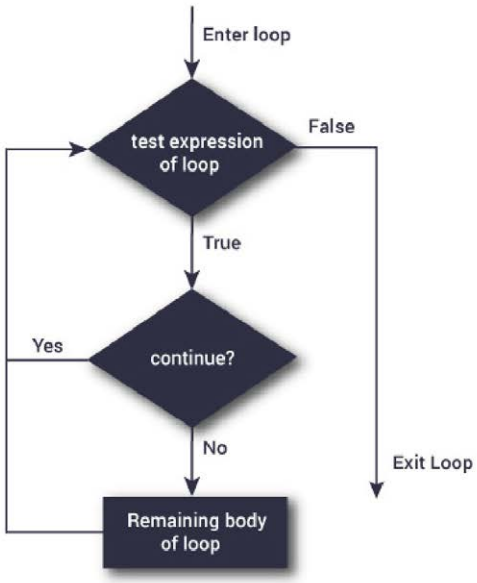
source: *https://bit.ly/2XE6MgD*

```
for var in sequence:
    # codes inside for loop
    if condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if condition:
        continue
    # codes  inside while loop

# codes outside while loop
```



- Example:

In [6]:
```
for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

```
s
t
r
n
g
The end
```

Lists

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples.

- How to create a list ?
  - Lists can be created by separating the list-items with comma.
  - List elements will be enclosed in (`[ ]`)

- For Example:

In [7]:
```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

In [9]:
```
for item in list1:
    print(type(item))
```

```
<class 'str'>
<class 'str'>
<class 'int'>
<class 'int'>
```

Indexing and Slicing

In [10]:
```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
```

In [11]:
```
# Get the first (index 0) of list1
list1[0]
```

Out[11]:
```
'physics'
```

In [13]:
```
## get the last element of list2
# Way 1
# ---- CODE HERE ----- #

# Way 2
# ---- CODE HERE ----- #
```

```
7
7
```

In [14]:
```
# Get the elements between index 1 (inclusive) up to 3 (exclusive) of li
st 1
list1[1:3] # notice that 3 is exclusive
```

Out[14]:
```
['chemistry', 1997]
```

In [16]:
```
# Get the elements of list2 starting from index 0 (inclusinve)
```

```
# up to index 5 (inclusinve) and with step 2
list2[0:6:2]
```

Out[16]: [1, 3, 5]

> Notice that the syntax is :
>
> ```
> list2[start:stop:step]
> ```
>
> Where the `start` index is inclusive and the `stop` index is exclusive.

In [17]:
```
# Get all the elemnts of list2 from index 1 to the end
list2[1:]
```

Out[17]: [2, 3, 4, 5, 6, 7]

In [18]:
```
# Get all the elemnts of list2 from index 1 to the end with step 2
list2[1::3]
```

Out[18]: [2, 5]

In [19]:
```
# Get all the elements of list2 backward
list2[::-1]
```

Out[19]: [7, 6, 5, 4, 3, 2, 1]

In [20]:
```
# Get the elements of list2 backward starting from the before last eleme
nt
list2[-2::-1]
```

Out[20]: [6, 5, 4, 3, 2, 1]

In [21]:
```
# Get the elements of list2 backward starting from the before last eleme
nt
# up to the element with index 2 (exclusive)
list2[-2:2:-1]
```

Out[21]: [6, 5, 4]

Updating a list

You can update single or multiple elements of lists. For example:

In [29]:
```
list = ['physics', 'chemistry', 1997, 2000];
print ("Value available at index 2 : ")
# ---- CODE HERE ----- #


print ("New value available at index 2 : ")
# ---- CODE HERE ----- #
```

```
Value available at index 2 :
```

```
1997
New value available at index 2 :
2001
```

Delete list elements

To remove a list element, you can use either the:

- `del` statement if you know exactly which element(s) you are deleting
- or `remove()` method if you do not know.

As example:

In [32]:
```
list1 = ['physics', 'chemistry', 1997, 2000];
print (list1)
del (list1[2]);
print ("After deleting value at index 2 : ")
print (list1)
```

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

> Note that `remove()` will be discussed in a subsequent section in this notebook.

Built-in List functions and methods

List operations

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Python includes the following list functions

Description	Function
len(list)	Gives the total length of the list.
max(list)	Returns item from the list with max value.
min(list)	Returns item from the list with min value.
list(seq)	Converts a tuple into list.

- For Example:

In [33]:
```
list1 = range(5)
list2 = range(3,10)
```

In [38]:
```
len(list1)
```

Out[38]: 5

In [39]:
```
max(list1)
```

Out[39]: 4

In [40]:
```
min(list2)
```

Out[40]: 3

Python includes the following list methods

Methods	Description
list.append(obj)	Appends object obj to list
list.count(obj)	Returns count of how many times obj occurs in list
list.extend(seq)	Appends the contents of seq to list
list.index(obj)	Returns the lowest index in list that obj appears
list.insert(index, obj)	Inserts object obj into list at offset index
list.pop(obj = list[-1])	Removes and returns last object or obj from list
list.remove(obj)	Removes the first object obj from list
list.reverse()	Reverses objects of list in place

list.sort([func])	Sorts objects of list, use compare func if given

- For example:

`append()`

In [41]:
```
a = ["bee", "moth"]
print(a)
# ---- CODE HERE ----- #
print(a)
```

```
['bee', 'moth']
['bee', 'moth', 'ant']
```

`count()`

In [42]:
```
a = ["bee", "ant", "moth", "ant"]
# ---- CODE HERE ----- #
```

```
1
2
0
```

`extend()`

In [43]:
```
a = ["bee", "moth"]
print(a)
# ---- CODE HERE ----- #
print(a)
```

```
['bee', 'moth']
['bee', 'moth', 'ant', 'fly']
```

`index()`

The possible syntax of `index()`:

1. `index(obj)`
2. `index(obj, start)`
3. `index(obj, start, stop)`

In [45]:
```
a = ["bee", "ant", "moth", "ant"]
# ---- CODE HERE ----- #
```

```
1
1
```

> Note that `index()` will raise a `ValueError` exception if no match was found.

In [46]:
```
try:
    # ---- CODE HERE ----- #
except ValueError as e:
    print("No can found !")
    print(e)
```

```
No can found !
'cat' is not in list
```

`insert()`

In [47]:
```
a = ["bee", "moth"]
print(a)
a.insert(0, "ant")
print(a)
a.insert(2, "fly")
print(a)
```

```
['bee', 'moth']
['ant', 'bee', 'moth']
['ant', 'bee', 'fly', 'moth']
```

`pop`

In [48]:
```
# Example 1: No index specified
a = ["bee", "moth", "ant"]
print(a)
# ---- CODE HERE ----- #
print(a)
```

```
['bee', 'moth', 'ant']
['bee', 'moth']
```

In [49]:
```
# Example 2: Index specified
a = ["bee", "moth", "ant"]
print(a)
# ---- CODE HERE ----- #
print(a)
```

```
['bee', 'moth', 'ant']
['bee', 'ant']
```

`remove()`

In [51]:
```
a = ["bee", "moth", "ant", "moth"]
print(a)
# ---- CODE HERE ----- #
print(a)
```

```
['bee', 'moth', 'ant', 'moth']
['bee', 'ant', 'moth']
```

`reverse()`

In [52]:
```
a = [3,6,5,2,4,1]
# ---- CODE HERE ----- #
print(a)
```

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

In [53]:
```
a = ["bee", "wasp", "moth", "ant"]
# ---- CODE HERE ----- #
print(a)
```

```
['ant', 'moth', 'wasp', 'bee']
```

In [55]:
```
# Also remember that you can do:
print(a)
# ---- CODE HERE ----- #
```

```
['ant', 'moth', 'wasp', 'bee']
```

Out[55]:
```
['bee', 'wasp', 'moth', 'ant']
```

`sort()`

Syntax:

```
sort(key=None, reverse=False)
```

- `key` : Specifies a function of one argument that is used to extract a comparison key from each list element. The default value is `None` (compares the elements directly).
- `reverse` : Boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

In [56]:
```
a = [3,6,5,2,4,1]
# ---- CODE HERE ----- #
print(a)
```

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

In [57]:
```
a = [3,6,5,2,4,1]
# ---- CODE HERE ----- #
print(a)
```

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

In [58]:
```
a = ["bee", "wasp", "moth", "ant"]
a.sort()
print(a)
```

```
['ant', 'bee', 'moth', 'wasp']
```

In [59]:
```
a = ["bee", "wasp", "butterfly"]
# ---- CODE HERE ----- #
print(a)
```

```
['bee', 'wasp', 'butterfly']
```

In [60]:
```
a = ["bee", "wasp", "butterfly"]
# ---- CODE HERE ----- #
print(a)
```

```
['butterfly', 'wasp', 'bee']
```

Tuples

The main difference between tuples and lists is that lists are mutable and tuples are not. A mutable object is one that can be changed. An **immutable** object is one that contains a fixed value that cannot be changed. If it needs to be changed, a new object must be created.

Creating a tuple

Creating a tuple is just like creating a list, except that you use regular brackets instead of square brackets.

For Example:

In [61]:
```
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
```

Actually, the brackets are optional (but it's always advised to keep your code readable - Better to use the brackets !). So you could also do this:

In [62]:
```
weekdays = "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"
```

In [63]:
```
print(weekdays)
print(type(weekdays))
```

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
<class 'tuple'>
```

Empty Tuple

In [64]:
```
tup=()
```

Single element tuple

> WARNING : The following may look bizarre, but you will get used to it

In [66]:
```
# ---- CODE HERE ----- #

print(type(a))
print(type(b))
```

```
<class 'str'>
<class 'tuple'>
```

Tuple containing a list

In [70]:
```
t = ("Banana", [1, 2, 3])
# Print the tuple
print(t)
```

```
('Banana', [1, 2, 3])
```

In [71]:
```
t[1].append(4)
```

In [72]:
```
print(t)
```

```
('Banana', [1, 2, 3, 4])
```

> Note : Ooops, we just broke the imutability of tuples ! Thanks to lists !)

Accessing tuple items

The same indexing that was used in list can be applied with tuples, for example:

In [74]:
```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )

# ---- CODE HERE ----- #
```

```
tup1[0]:  physics
tup2[1:5]:  (2, 3, 4, 5)
```

In [75]:
```
t = (101, 202, ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
)
# ---- CODE HERE ----- #
```

```
Tuesday
```

Updating tuples

Tuples are immutable, which means you cannot update or change the values of tuple elements. You are able to take portions of the existing tuples to create new tuples as the following example :

In [77]:
```
import sys # To enable showing general exception info
```

In [82]:
```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

try:
```

```
    # Following action is not valid for tuples
    tup1[0] = 100
except:
    e_type, e_msg = sys.exc_info()[:2]
    print("Error")
    print("\t", e_type)
    print("\t", e_msg)
    print()
    pass

# So let's create a new tuple as follows (concatenating tuples)
tup3 = tup1 + tup2
print (tup3)
```

```
Error
        <class 'TypeError'>
        'tuple' object does not support item assignment

(12, 34.56, 'abc', 'xyz')
```

Deleting a tuple

In [85]:
```
tup = ('physics', 'chemistry', 1997, 2000);

print (tup)
del tup

print ("After deleting tup : ")
try:
    print (tup)
except:
    e_type, e_msg = sys.exc_info()[:2]
    print()
    print("Error")
    print("\t", e_type)
    print("\t", e_msg)
    print()
    pass
```

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :

Error
        <class 'NameError'>
        name 'tup' is not defined
```

Tuple build-in functions

In [88]:
```
tuple1, tuple2 = (123, 'xyz'), (456, 20)
```

```
print(len(tuple1))
print(max(tuple2))
```

```
2
456
```

Dictionaries

Dictionaries are pairs of **`key`**`:`**`value`** mutable lists. A dictionary is created using (`{ }`).

Creating a dictionary

Empty dictionary

In [93]:
```
d = {}
print(type(d))
```

```
<class 'dict'>
```

- The content of the dictionary can be updated:

In [92]:
```
d['key1'] = 'value1'
d[2] = 'two'

print(d)
```

```
{'key1': 'value1', 2: 'two'}
```

Simple creation of dictionary

In [94]:
```
d = {"Key1": "Value1", "Key2": "Value2"}
print(d)
```

```
{'Key1': 'Value1', 'Key2': 'Value2'}
```

Creating a dictionary from tuples

In [97]:
```
d1 = dict([("Earth", 40075), ("Saturn", 378675), ("Jupiter", 439264)])
d2 = dict(Earth=40075, Saturn=378675, Jupiter=439264)
print(d1)
print(type(d1))
print()
print(d2)
print(type(d2))
```

```
{'Earth': 40075, 'Saturn': 378675, 'Jupiter': 439264}
<class 'dict'>
```

```
{'Earth': 40075, 'Saturn': 378675, 'Jupiter': 439264}
<class 'dict'>
```

Dictionary containing lists and tuples

In [98]:
```
# Create the dictionary
food = {"Fruit": ["Apple", "Orange", "Banana"], "Vegetables": ("Eat", "
Your", "Greens")}
# Print the dictionary
print(food)
```

```
{'Fruit': ['Apple', 'Orange', 'Banana'], 'Vegetables': ('Eat', 'Your', '
Greens')}
```

Accessing dictionary

In [99]:
```
# Create the dictionary
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
```

Getting dictionary keys

In [100]:
```
print(planet_size.keys())
```

```
dict_keys(['Earth', 'Saturn', 'Jupiter'])
```

Getting dictionary values

In [101]:
```
print(planet_size.values())
```

```
dict_values([40075, 378675, 439264])
```

Accessing dictionary values through its key

Using `[]`

In [102]:
```
# ---- CODE HERE ----- #
```

Out[102]:
```
378675
```

Trying to index non existing key using this method result and `KeyError` exception.

In [104]:
```
import sys
```

In [105]:
```
try:
    planet_size['Pluto']
except:
    e_type, e_msg = sys.exc_info()[:2]
    print("ERROR :")
    print("\t Typle : ",e_type)
```

```
    print("\t Message : ",e_msg)
```

```
ERROR :
        Typle :  <class 'KeyError'>
        Message :  'Pluto'
```

Using `get()`

In [103]:
```
# ---- CODE HERE ----- #
```

Out[103]:
```
378675
```

Trying to fetch non existing key using `get()` will return the value `None`.

In [107]:
```
# ---- CODE HERE ----- #
```

```
None
```

Also you can set a default value that would be returned instead on `None` if no key match was found.

In [108]:
```
# ---- CODE HERE ----- #
```

```
I think 1188
```

Updating Dictionary

You can update the dictionary by referring to the item's key. If the key exists, it will update its value to the new value. If the key doesn't exist, it will create a new *key:value* pair.

In [113]:
```
# Create the dictionary and print
user = {"Name": "Christine", "Age": 23}
print(user)
# Update the user's age and print

# ---- CODE HERE ----- #

print(user)
```

```
{'Name': 'Christine', 'Age': 23}
{'Name': 'Christine', 'Age': 24}
```

In [114]:
```
# Add a new item and print

# ---- CODE HERE ----- #

print(user)
```

```
{'Name': 'Christine', 'Age': 24, 'Height': 154}
```

In [116]:
```
# Deleting the age

# ---- CODE HERE ----- #

```

```
print(user)
```

```
{'Name': 'Christine', 'Height': 154}
```

Deleting the whole dictionary

In [117]:
```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

# ---- CODE HERE ----- #
```

Dictionary built-in methods

Method	Description
dict.clear()	Removes all elements of dictionary *dict*
dict.copy()	Returns a shallow copy of dictionary *dict*
dict.fromkeys()	Create a new dictionary with keys from seq and values *set* to *value*.
dict.get(key, default=None)	For *key* key, returns value or default if key not in dictionary
dict.has_key(key)	Removed, use the *in* operation instead.
ict.items()	Returns a list of *dict*'s (key, value) tuple pairs
dict.keys()	Returns list of dictionary dict's keys
dict.setdefault(key, default = None)	Similar to get(), but will set dict[key] = default if *key* is not already in dict
dict.update(dict2)	Adds dictionary *dict2*'s key-values pairs to *dict*
dict.values()	Returns list of dictionary *dict*'s values

Functions

A function is a *named* block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python Built-in function

Method	Description
abs()	returns absolute value of a number
all()	returns true when all elements in iterable is true
any()	Checks if any Element of an Iterable is True
ascii()	Returns String Containing Printable Representation
bin()	converts integer to binary string
bool()	Converts a Value to Boolean
bytearray()	returns array of given byte size
bytes()	returns immutable bytes object
callable()	Checks if the Object is Callable
chr()	Returns a Character (a string) from an Integer
classmethod()	returns class method for given function
compile()	Returns a Python code object
complex()	Creates a Complex Number
delattr()	Deletes Attribute From the Object
dict()	Creates a Dictionary
dir()	Tries to Return Attributes of Object
divmod()	Returns a Tuple of Quotient and Remainder
enumerate()	Returns an Enumerate Object
eval()	Runs Python Code Within Program
exec()	Executes Dynamically Created Program
filter()	constructs iterator from elements which are true
float()	returns floating point number from number, string
format()	returns formatted representation of a value
frozenset()	returns immutable frozenset object
getattr()	returns value of named attribute of an object
globals()	returns dictionary of current global symbol table
hasattr()	returns whether object has named attribute
hash()	returns hash value of an object
help()	Invokes the built-in Help System
hex()	Converts to Integer to Hexadecimal
id()	Returns Identify of an Object
input()	reads and returns a line of string
int()	returns integer from a number or string
isinstance()	Checks if a Object is an Instance of Class
issubclass()	Checks if a Object is Subclass of a Class

iter()	returns iterator for an object
len()	Returns Length of an Object
list() Function	creates list in Python
locals()	Returns dictionary of a current local symbol table
map()	Applies Function and Returns a List
max()	returns largest element
memoryview()	returns memory view of an argument
min()	returns smallest element
next()	Retrieves Next Element from Iterator
object()	Creates a Featureless Object
oct()	converts integer to octal
open()	Returns a File object
ord()	returns Unicode code point for Unicode character
pow()	returns x to the power of y
print()	Prints the Given Object
property()	returns a property attribute
range()	return sequence of integers between start and stop
repr()	returns printable representation of an object
reversed()	returns reversed iterator of a sequence
round()	rounds a floating point number to ndigits places.
set()	returns a Python set
setattr()	sets value of an attribute of object
slice()	creates a slice object specified by range()
sorted()	returns sorted list from a given iterable
staticmethod()	creates static method from a function
str()	returns informal representation of an object
sum()	Add items of an Iterable
super()	Allow you to Refer Parent Class by super
tuple() Function	Creates a Tuple
type()	Returns Type of an Object
vars()	Returns __dict__ attribute of a class
zip()	Returns an Iterator of Tuples
__import__()	Advanced Function Called by import

source: *https://bit.ly/2Qou3At*

User-defined functions function

Syntax:

```
def function_name(arg_1, arg_2, ...):
    """ docstring """
    # Function body
    return expression
```

- Function definition must contain `def`, function name, `()` and `:`
- Arguments are optional (_arg*1*, _arg*2*)
- Docstring is optional, and it contains a description about the aim of the function, its inputs and its outputs
- `return` is optional if the function returns nothing.

Example : Square the elements of a numeric lists

- Creating a function

In [118]:
```
import numpy as np
```

In [119]:
```
l = np.arange(10)
print(l)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

In [124]:
```
# Method 1 :  Creating a function
def my_square(in_list):
    """my_square creats a return the square
    for each element in passed numeric list

    inputs :
        + in_list : numeric list
    outputs :
        square of each element in in_list
    """

# ---- CODE HERE ----- #
```

In [125]:
```
# Call my_square function
my_square(l)
```

Out[125]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [127]:
```
# Method 2

# ---- CODE HERE ----- #

result
```

Out[127]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- To get the information about `my_square` method:

In [128]:
```
help(my_square) # In native pyhton
```

```
Help on function my_square in module __main__:

my_square(in_list)
    my_square creats a return the square
    for each element in passed numeric list

    inputs :
        + in_list : numeric list
    outputs :
        square of each element in in_list
```

In [129]:
```
my_square? # In IPython
```

Returning multiple values

You can return multiple values from a function by separating them with a comma. This actually creates a tuple, which you can use as is, or unpack in order to use the individual values.

For Example:

In [132]:
```
# Function to perform the four basic arithmetic operations
# against two numbers that are passed in as arguments.
def basic_arithmetic(x, y):


# ---- CODE HERE ----- #
```

In [137]:
```
# Call the function and print the result
comput_result = basic_arithmetic(3, 30)
print(comput_result)
```

```
(33, 90, 0.1, -27)
```

In [135]:
```
# Call the funtion again and save the results in seperate variables
r_sum, r_prod, r_div, r_diff = basic_arithmetic(3,30)
print("sum : ", r_sum)    # comput_result[0]
print("prod : ", r_prod)  # comput_result[1]
print("div : ", r_div)    # comput_result[2]
print("diff : ", r_diff)  # comput_result[3]
```

```
sum :  33
prod :  90
div :  0.1
diff :  -27
```

In [136]:
```
# If you are interested only in the product
_, r_prod, _, _ = basic_arithmetic(3,30)
print("prod : ", r_prod)
```

```
prod :  90
```

Function Input Arguments

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

For Example:

In [138]:
```
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return
```

In [142]:
```
printinfo( age = 50, name = "miki" )
print()

printinfo( name = "miki" )
print()

printinfo('miki')
print()

try:
    printinfo(age=50) # name is a required argument
except:
    e_type, e_msg = sys.exc_info()[:2]
    print("Error : printinfo(age=50)")
    print("\t+ ",e_type)
    print("\t+ ",e_msg)
```

```
Name:  miki
Age  50

Name:  miki
Age  35

Name:  miki
Age  35

Error : printinfo(age=50)
        +  <class 'TypeError'>
        +  printinfo() missing 1 required positional argument: 'name'
```

> **Note :** Default arguments **must** be on the most-right of the function input arguments list.

Arbitrary arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

To achieve this, in the function definition, precede the *arbitrary argument* with `*`. For Example:

In [143]:
```
def greet(*names):
    """This function greets all
    the person in the names tuple."""



# ---- CODE HERE ----- #
```

In [144]:
```
greet("Monica","Luke","Steve","John")
```

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

In [145]:
```
greet()
```

```
Is anybody out there ?
```

`*args` and `**kwargs`

`*args` and `**kwargs` are idioms usually used to declare *Arbitrary Arguments*, where both allow passing variable number of arguments to a function.

- `*args` : idiom for **non-keyworded** variable-length argument (like `*names` in the previous example)
- `**kwargs` : idiom for keyworded variable-length argument. You should use `**kwargs` if you want to handle named arguments in a function.

Example for `*args` (again) :

In [163]:
```
def greet(prefix, *args):
    """
    This function greets all
    the person in the names tuple.

    + prefix can be either Mr. or Mrs. or Miss.
    """
    allowed_prefix = ['mr.', 'mrs.', 'miss.']


# ---- CODE HERE ----- #
```

In [159]:
```
greet('Mr.', 'Mike')
```

```
Hello Mr. Mike
```

In [161]:
```
greet('Mr.', 'Mike', 'Roland')
```

```
Hello Mr. Mike
Hello Mr. Roland
```

In [165]:
```
greet('Miss.')
```

```
Hello Miss. Anonymous
```

Example for `**kwargs`:

In [169]:
```
def greet_me(**kwargs):
    if kwargs is not None:
        print("Type of kwargs : ", type(kwargs)) # to proove that kwargs is a dict
        print()
        for key, value in kwargs.items():
            print ("%s == %s" %(key,value))
```

In [170]:
```
greet_me(name="Ghassan")
```

```
Type of kwargs :  <class 'dict'>

name == Ghassan
```

Anonymous/Lambda Functions

While normal functions are defined using the `def` keyword, in Python anonymous functions are defined using the `lambda` keyword. Therefore anonymous functions are also called `lambda` functions.

Syntax

```
lambda arguments: expression
```

Here is an example of lambda function that doubles the input value:

In [171]:
```
# Program to show the use of lambda functions
double = lambda x: x * 2

# Output: 10
print(double(5))
```

```
10
```

Also lambda functions can take multiple inputs, as example:

In [172]:
```
mal = lambda x,y : x*y

print(mal(5,6))
```

```
30
```

Lambda with **filter()**

The `filter()` function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluats to True.

In [173]:
```
filter?
```

Exercise

Given the following 2D list (`mat`), use `filter` and `lambda` function to retrieve the even elements.

In [ ]:
```
mat = [[1,2,3,4],
       [5,6,7,8]]

### YOUR CODE HERE
#
#
###
```

Lambda with **map()**

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

For example:

In [9]:
```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

# Output: [2, 10, 8, 12, 16, 22, 6, 24]
print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Exercise

Given the following 2D list (`mat`), use `map` and `lambda` function to scale every element by subtracting the mean of the row where the element exists from it.

In [ ]:
```
mat = [[1,2,3,4],
       [5,6,7,8]]

### YOUR CODE HERE
#
#
###
```

`assert`

Assertions are statements that assert or state a fact confidently in your program.

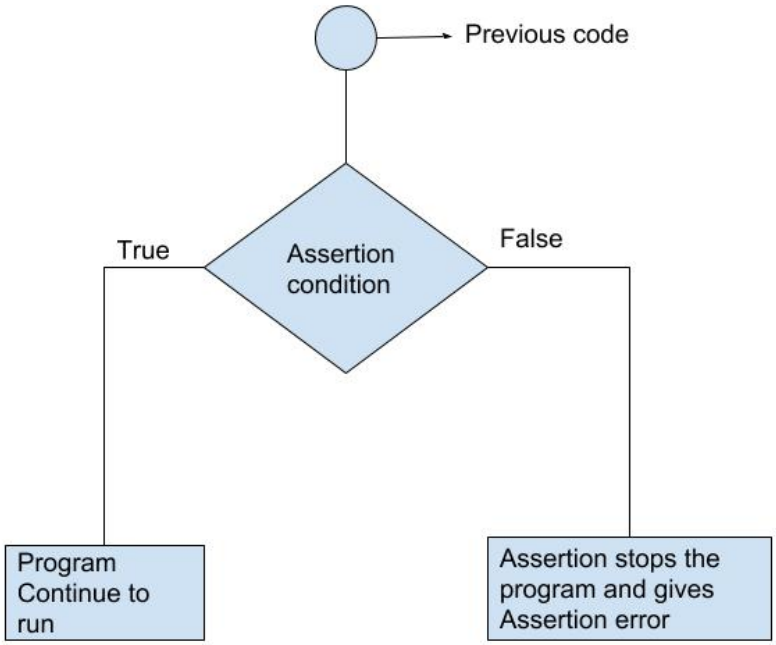
Assertions are simply boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and move to the next line of code. However, if it's false, the program stops and throws an error.

Syntax

```
pyton
assert <condition>

assert <condition>,<error message>
```

source: https://bit.ly/2kfDaJ4



- Example:

In [22]:
```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

```
Average of mark2: 78.0

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-22-4a885bedd40a> in <module>
      7
      8 mark1 = []
----> 9 print("Average of mark1:",avg(mark1))

<ipython-input-22-4a885bedd40a> in avg(marks)
      1 def avg(marks):
----> 2     assert len(marks) != 0,"List is empty."
      3     return sum(marks)/len(marks)
      4
      5 mark2 = [55,88,78,90,79]

AssertionError: List is empty.
```

References

[1] **[BLOG]** The Zen of Python.

[2] **[BOOK]** Python for Data Analysis: A Quick Python Learning Guide for Beginners, L.Malik and S.Arora.

[3] **[BOOK]** Python for Data Analysis, Wes McKinney

[4] **[BOOK]** Python Cookbook, D.Beazley, B.K.Jones

[5] **[BOOK]** Automate The Boring Stuff With Python, A.Sweigart