# Python Lecture 3 – Programming

- Flow control instructions
- Files
- Practice

## Bibliography and learning materials

- ★ Bibliography:

  https://www.python.org/doc/

  http://docs.python.it/

  and much more available in internet

- ★ Learning Materials:

  https://github.com/gtaffoni/Learn-Python/tree/master/Lectures

  https://github.com/bertocco/bash_lectures

# The if statement

The **if** statement is used for conditional execution: if a condition is true, we run a block of statements (called the if-block), else we process another block of statements (called the else-block).

The else clause is optional.

**Syntax:**

```
if test_expression :
    statement(s)
```

or

```
if test_expression :
    body of if
else:
    body of else
```

```
if test_expression1 :
    body of if
elif test_expression2 :
    body of elif
else:
    body of else
```

switch-case simulation

## The if statement: example

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here

print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

# The while statement

The **while** statement allows you to repeatedly execute a block of statements as long as a condition is true.
A while statement is an example of what is called a looping statement.
A while statement can have an optional else clause. If the else clause is present, it is always executed once after the while loop is over unless a break statement is encountered.

**Syntax:**

```
while test_condition :
    while-statement(s)
[else:
    else-statement(s)]
```

else clause is optional

# The while statement: example

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here
print('Done')
```

## The for statement

The **for** statement is a looping statement which iterates over a sequence of objects, i.e. go through each item in a sequence. A sequence is just an ordered collection of items.

In general we can use any kind of sequence of any kind of objects.

An else clause is optional, when included, it is always executed once after the for loop is over unless a break statement is encountered.

**Syntax:**

```
for iterating_var in sequence:
    statements(s)
[else:
    else-statement(s)]
```

else clause is optional

**Example:**

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

## The break statement

The **break** statement is used to break out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has not been completely iterated over.

An important note is that if you break out of a for or while loop, any corresponding loop else block is not executed.

**Example** (break.py)**:**

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

## Exercise

Try a for and a while loop with an else clause verifying that the else clause is always executed except in case a break statement is found.

Solutions (in https://github.com/bertocco/bash_lectures/):

examples/python_3/while_break.py

examples/python_3/for_break.py

## The continue statement

The **continue** statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

**Example** (continue.py)**:**

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
    print('Input is of sufficient length')
    # Do other kinds of processing here…
```

=> the continue statement works with the for loop as well.

# The pass statement

The **pass** statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

**Example**:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
```

- This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
```

- Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
>>> def initlog():
...     pass   # Remember to implement this!
```

## Exercise: control_flow_step0

Prepare a python script where all the presented examples on flow control statements are converted in functions.

Write a main block of code printing instructions and explanations useful to the user and then calling the functions.

Example of expected output:

```
This is if statement usage example.
You have to guess the right number trying repetitively:
Enter an integer :
…….
This is while statement usage example.
…….
```

*and so on…...*

# Exercise: control_flow_step1

Complicate the previous script giving the user the ability to choose which statement he likes to try.

Output example:

```
Choose if try
1. if statement
2. while statement
3. for statement
make your choose entering the number (1 or 2 or 3)
……..
```

## Exercise: control_flow_step2

Complicate the previous script giving the user the ability to choose how much iteration execute in case it is trying the for statement

Output example:

```
Choose if try
1. if statement
2. while statement
3. for statement
make your choose entering the number (1 or 2 or 3)
3
Enter how much iteration you want execute (integer)
```

## Exercise: control_flow_step3

Complicate the previous script giving the user the ability to choose repeatedly the control statement to test.

## Exercise: control_flow_step4

Complicate one of the previous scripts giving the user the ability to choose the reference number used for comparison in if and while statements (fixed to guess=23 in the already done exercices).

## Iterators

**for** cicle is generally used to iterate on iterable types like list, tuple, string, and in general containers.

Iterables types contain an object called iterator used by the for operator to iterate in the container.

The iterator object contains a next() method, returning the first available data in the container, useful to iterate in the container.

## Iterators. examples

```
>>> a = iter(list(range(10)))

>>> for i in a:

...     print(i)

...     next(a)

...

0

1

2

3

4

5

6

7

8
```

```
>>> a = iter(list(range(10)))
>>> for i in a:
...     next(a)
...
1
3
5
7
9
```

```
>>> for i in a:
...     print("Printing: %s" % i)
...     next(a)
...
Printing: 0
1
Printing: 2
3
Printing: 4
5
Printing: 6
7
Printing: 8
9
>>>
```

## Data sequences and cicles

**for** cicle allows to iterate on every kind of iterable object like list, tuple, string, set, dictionary.

**Example:**

**LIST**

```
>>> a=[1,2,3,4,5]
>>> for el in a:
            print el
1
2
3
4
5
```

**STRING**

```
>>> a='"Ciao"'
>>> for el in a:
            print el
C
i
a
o
```

**SET**

```
>>>a=set([1,2,3,4])
>>> for el in a:
            print el
1
2
3
4
```

# Data sequences and cicles

**for** cicle allows to iterate on every kind of iterable object like list, tuple, string, set, dictionary.

**Example:**

**DICTIONARY (by key)**

```
>>> a={1:'a',2:'b'}
>>> for el in a.keys():
        print el
1
2
```

**DICTIONARY(by value)**

```
>>> a={1:'a',2:'b'}
>>> for el in a.values():
        print el
a
b
```

**DICTIONARY(by key-val)**

```
>>> a={1:'a',2:'b'}
>>> for k,v in a.items():
        print k,v
1 a
2 b
```

**DICTIONARY**

```
>>> a={1:'a',2:'b'}
>>> for el in a:
        print el
1
2
```

**DICTIONARY**

```
>>> a={1:'a',2:'b'}
>>> for el in (1,2,3):
        print a.get(el)
a
b
None
```

# range() and xrange() functions

range() and xrange() are functions for creating lists, or a range of integers that assist in making for loops.
For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use, however you please.
The only difference is that
=> range returns a Python list object
=> xrange returns an xrange object.
It means that xrange doesn't actually generate a static list at run-time like range does. It creates the values as you need them.
That means that if you have a really gigantic range you'd like to generate a list for, say one billion, xrange is the function to use. This is especially true if you have a really memory sensitive system such as a cell phone that you are working with, as range will use as much memory as it can to create your array of integers, which can result in a MemoryError and crash your program. It's a memory hungry beast.
If you'd like to iterate over the list multiple times, it's probably better to use range. This is because xrange has to generate an integer object every time you access an index, whereas range is a static list and the integers are already "there" to use.

## range() and xrange() functions

The functions xrange and range take in three arguments in total, however two of them are optional. The arguments are "start", "stop" and "step". "start" is what integer you'd like to start your list with, "stop" is what integer you'd like your list to stop at, and "step" is what your list elements will increment by.

```
>>> for i in xrange(1, 10, 2):
...     print(i)
...
1
3
5
7
9
```

Python's xrange and range with Negative Numbers:

```
>>> for i in xrange(-1, -10, -1):
...     print(i)
...
-1
-2
-3
-4
-5
-6
-7
-8
-9
```

*you must do it this way for negative lists. Trying to use xrange(-10) will not work because xrange and range use a default "step" of one.*

Note that if "start" is larger than "stop", the list returned will be empty. Also, if "step" is larger that "stop" minus "start", then "stop" will be raised to the value of "step" and the list will contain "start" as its only element.

**Example**:

```
>>> for i in xrange(70, 60):
...         print(i)
...
# Nothing is printed
>>> for i in xrange(10, 60, 70):
...         print(i)
...
10
```

**Syntax**:

```
xrange(stop)
xrange(start, stop[, step])
range(stop)
range(start, stop[, step])
```

## Deprecation of Python's xrange in Python 3

Python 3.x's range function is xrange from Python 2.x.

In Python 3.x, the xrange function does not exist anymore.

The range function now does what xrange does in Python 2.x

To keep your code portable, you might want to stick to using range instead.

The reason why xrange was removed was because it is basically always better to use it, and the performance effects are negligible.

# Exceptions

Errors detected during execution are called exceptions.

Exceptions are errors raised executing a statement or an expression, also in case they are syntactically correct.

Exceptions are not unconditionally fatal: they can be handled in Python programs. Most exceptions are not handled by programs, however, and result in error messages.

**Example**:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- Exceptions come in different types, and the type is printed as part of the message. Example are ZeroDivisionError, NameError and TypeError.

## Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

**Example:**

- Create class

```
class MyClass:
  def __init__(self, name, age):
    self.attribute1 = value1
    self.attribute2 = value2

  def myfunc(self):
    print("Hello my attrib1 is " + self.attribute1)
```

**Example:**

- Create and use object
-

```
p1 = MyClass()

p1.myfunc()
print(p1.attribute1)
```

## Classes: the __init__ object

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated, i.e.every time the class is being used to create a new object.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

**Example**

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

## Classes: methods

Classes can also contain methods. Methods in objects are functions that belongs to the object.

Let us create a method in the Person class  that prints a greeting, and execute it on the p1 object:

**Example**

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

## User Defined Exceptions

Programs may name their own exceptions by creating a new exception class.

# Exceptions handling

Programs can handle exceptions with the following structure

**try:**

    statement(s)

**except** ExceptionType1**:**

    statement(s)

**except** exceptionType2, exceptionType3:

    statement(s)

……

**except:**

    statement(s)

**else:**

    statement(s)

**finally:**

    statement(s)

except EceptionType1 is executed if an Exception of Type1 is raised in the try block

except is executed if a not previously catchd exception is thrown

else is executed if no one exception is thrown in try block

finally is always executed

## Exceptions handling: try….except clause

The **try** statement works as follows: the try clause (the statement(s) between the try and except keywords) is executed.

If no exception occurs, the except clause is skipped and the execution of the try statement is finished.

If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the **except** clause is executed, and then execution continues after the try statement.

If an exception occurs which does not match the exception named in the except clause, it is passed on to other except statements and at the end, to the generic except clause, if it is present. If no handler is found, it is an unhandled exception and execution stops with a message.

When a try statement has more than one except clause, to specify handlers for different exceptions, at most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.

## Exceptions handling: except clause

An **except** clause may name multiple exceptions as a parenthesized tuple. Example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

# Exceptions handling: else clause

The try … except statement has an optional **else** clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. **Example**:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

## Exceptions handling: final clause

The try statement has the **final** optional clause.

The final clause,  which is intended to define clean-up actions,
is always executed before leaving the try statement, whether an exception has occurred or not.

When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in an except or else clause), it is re-raised after the finally clause has been executed.

The finally clause is also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement.

# Exceptions handling: a complete example

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

## Exceptions handling: the raise statement

The **raise** statement allows the programmer to force a specified exception to occur. For example:

```
>>>
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

# Read text file in matrix

Input1.txt:

```
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,2,1,0,2,0,0,0,0
0,0,2,1,1,2,2,0,0,1
0,0,1,2,2,1,1,0,0,2
1,0,1,1,1,2,1,0,2,1
```

Code to read file:

```
with open('Input2.txt', 'r') as f:
    data = f.readlines() # read raw lines into an array

cleaned_matrix = []
for raw_line in data:
    split_line = raw_line.strip().split(",") # ["1", "0" ... ]
    nums_ls = [int(x.replace('"', '')) for x in split_line]
    # get rid of the quotation marks and convert to int
    cleaned_matrix.append(nums_ls)

print cleaned_matrix
```

# Read text file in matrix

Input1.txt:

```
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,2,1,0,2,0,0,0,0
0,0,2,1,1,2,2,0,0,1
0,0,1,2,2,1,1,0,0,2
1,0,1,1,1,2,1,0,2,1
```

Code to read file:

```
fin = open('input.txt','r')
a=[]
for line in fin.readlines():
    a.append( [ int (x) for x in line.split(',') ] )
```

```
l = []
with open('input.txt', 'r') as f:
  for line in f:
    line = line.strip()
    if len(line) > 0:
      l.append(map(int, line.split(',')))
print l
```

```
fin = open('input.txt','r')
a=[]
for line in fin.readlines():
    a.append( [ int (x) for x in line.split(',') ] )
```

## Read text file in matrix using numpy

Input1.txt:

```
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,2,1,0,2,0,0,0,0
0,0,2,1,1,2,2,0,0,1
0,0,1,2,2,1,1,0,0,2
1,0,1,1,1,2,1,0,2,1
```

Code to read file:

```
import numpy as np
input = np.loadtxt("input.txt", dtype='i',
delimiter=',')
print(input)
```

numpy is a library

numpy.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max_rows=None) [source]
Load data from a text file.

Each row in the text file must have the same number of values.

https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

## Read text file in matrix: example

Input2.txt:

```
"0","0","0","0","1","0"
"0","0","0","2","1","0"
```

Code to read file:

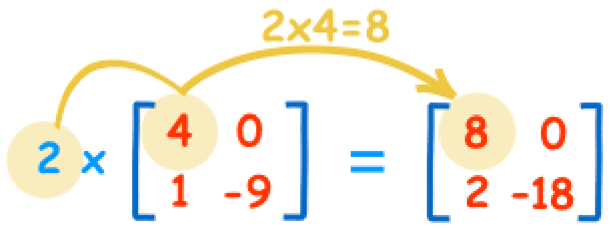
```
with open('Input2.txt', 'r') as f:
    data = f.readlines() # read raw lines into an array

cleaned_matrix = []
for raw_line in data:
    split_line = raw_line.strip().split(",") # ["1", "0" ... ]
    nums_ls = [int(x.replace('"', '')) for x in split_line] # get rid of the
quotation marks and convert to int
    cleaned_matrix.append(nums_ls)

print cleaned_matrix
```

# Multiply matrices: Matrix Multiply Constant

To multiply a matrix by a single number is easy:



$$2 \times \begin{bmatrix} 4 & 0 \\ 1 & -9 \end{bmatrix} = \begin{bmatrix} 8 & 0 \\ 2 & -18 \end{bmatrix}$$

These are the calculations:
2×4=8  2×0=0
2×1=2  2×-9=-18

We call the number ("2" in this case) a scalar, so this is called "scalar multiplication".

https://www.mathsisfun.com/algebra/matrix-multiplying.html

# Multiply matrices: Multiplying a Matrix by Another Matrix

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$

1st row X 1st column:
(1, 2, 3) • (7, 9, 11) = 1×7 + 2×9 + 3×11
= 58

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ & \end{bmatrix}$$

1st row X 2nd column:
(1, 2, 3) • (8, 10, 12) = 1×8 + 2×10 + 3×12
= 64

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix} \checkmark$$

2nd row X 1st column:
(4, 5, 6) • (7, 9, 11) = 4×7 + 5×9 + 6×11
= 139

2nd row X 2nd column:
(4, 5, 6) • (8, 10, 12) = 4×8 + 5×10 + 6×12
= 154

Matrix product is possible only between matrices
nXm mXp → nXp (result dimension)

https://www.mathsisfun.com/algebra/matrix-multiplying.html

## Example

```
# Program to multiply two matrices using nested loops
# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
         [0,0,0,0],
         [0,0,0,0]]
# iterate through rows of X
for i in range(len(X)):
   # iterate through columns of Y
   for j in range(len(Y[0])):
       # iterate through rows of Y
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]
for r in result:
   print(r)
```

**Output:**

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

## Example Matrix Multiplication Using Nested List Comprehension

```
# Program to multiply two matrices using list comprehension

# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]

# result is 3x4
result = [[sum(a*b for a,b in zip(X_row,Y_col)) for Y_col in zip(*Y)] for X_row in X]

for r in result:
   print(r)
```

**Output:**

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

## Exercise

Write a python script to multiply two matrices.
You can use the previous example.
The matrices can be defined inside the program or read by file.
Try the case in which matrices are in two different files or in one unique file.

Try also the special case of product between matrix and vector
[mXn X nX1]

Verify with an example that
AXB != BXA [must be mXn * nXm]
Suggestion: incapsulate the matrix product in a function receiving the two matrices as parameters.