Python Lecture 3
Control Flow and Python Programming

- Flow control instructions
- Iterators
- Simple matrix examples
- Lambda expressions
- Input

S. Bertocco

Bibliography and learning materials

★ Bibliography:

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

http://docs.python.it/

https://www.w3schools.com/python/

https://pynative.com/python-exercises-with-solutions/

and much more available in internet

★ Learning Materials:

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

https://github.com/bertocco/abilita_info_units_2021

Control flow instructions

- Decision
- Cycle

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')
```

“do … until” does not exist in Python

The do … until cycle in python does not exist.
It can be emulated. Examples:

```
while True:
    do_something()
    if condition():
       break
```

or:

```
finished = False
while not finished:
    ... do something...
    finished = evaluate_end_condition()
```

https://stackoverflow.com/questions/1662161/is-there-a-do-until-in-python

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.

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:

```
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.

Iterable 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)

0

1

2

3

4

5

6

7

8

9
```

```
>>> 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() function

Syntax:

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

The range() function returns a sequence of numbers,
starting from start (= 0 by default),
increments by step (= 1 by default),
stops before stop, range() function doesn't include the last (stop) number in the result.

range() function: examples

Generic with 3 parameters:

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

With Negative Numbers:

```
>>> for i in range(-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 range(-10) will not work because range uses 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 range(70, 60):
...         print(i)
...
# Nothing is printed
>>> for i in range(10, 60, 70):
...         print(i)
...
10
```

range() function: Exercises

Create a sequence of numbers from 3 to 5, and print each item in the sequence

```
x = range(3, 6)
for n in x:
    print(n)
```

Create a sequence of numbers from 3 to 19, but increment by 2 instead of 1

```
x = range(3, 20, 2)
for n in x:
    print(n)
```

zip() function

Syntax:

zip(iterator1, iterator2, iterator3 ...)

The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.

If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

zip() function: example

```
a = ("Marco", "Luca", "Claudio")
b = ("Giovanna", "Maria", "Anna", "Francesca")

x = zip(a, b)

print(tuple(x))
```

Result:

```
(('Marco', 'Giovanna'), ('Luca', 'Maria'), ('Claudio', 'Anna'))
```

If one tuple contains more items, these items are ignored

zip() function: usage example

The * operator can be used in conjunction with zip() to unzip the list.

Example:

```
coordinate = ['x', 'y', 'z']
value = [3, 4, 5]

result = zip(coordinate, value)
result_list = list(result)
print(result_list)

c, v =  zip(*result_list)
print('c =', c)
print('v =', v)
```

Output:

```
[('x', 3), ('y', 4), ('z', 5)]
c = ('x', 'y', 'z')
v = (3, 4, 5)
```

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:

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

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(',') ] )
```

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

```
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

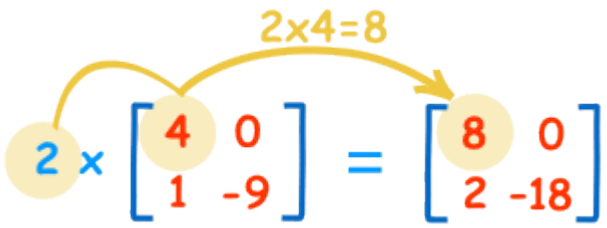
Input2.txt:
"0","0","0","0","1","0"
"0","0","0","2","1","0"

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:

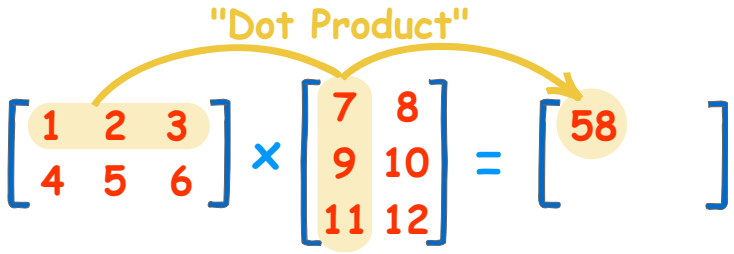


These are the calculations:
2×4=8 2×0=0
2×1=2 2×-9=-18

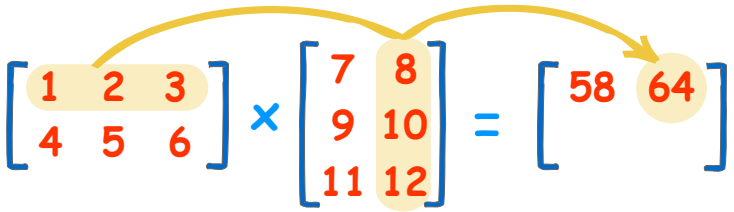
We call the number ("2" in this case) a scalar, so this is called "scalar multiplication".

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

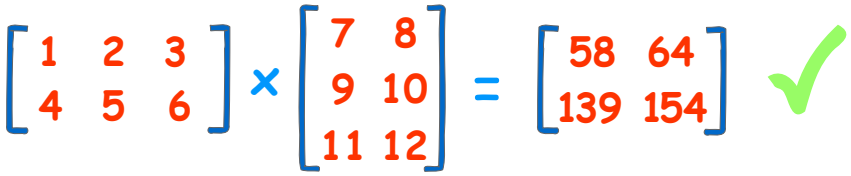
Multiply matrices: Multiplying a Matrix by Another Matrix



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



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



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]
```

zip() function

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and return it.

Syntax of the zip() function is:

zip(*iterables)

Return Value

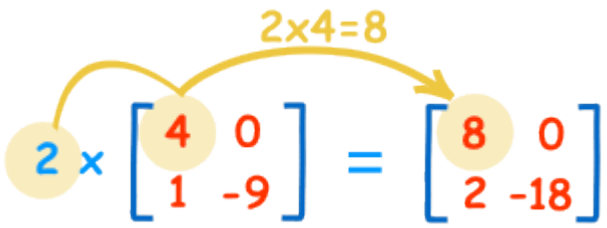
The zip() function returns an iterator of tuples based on the iterable objects.

If we do not pass any parameter, zip() returns an empty iterator
If a single iterable is passed, zip() returns an iterator of tuples with each tuple having only one element.
If multiple iterables are passed, zip() returns an iterator of tuples with each tuple having elements from all the iterables.

Example
Suppose, two iterables are passed to zip(); one iterable containing three and other containing five elements. Then, the returned iterator will contain three tuples. It's because iterator stops when the shortest iterable is exhausted.

Multiply matrices: Matrix Multiply Constant

To multiply a matrix by a single number is easy:



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

Exercise 1: matrix x scalar

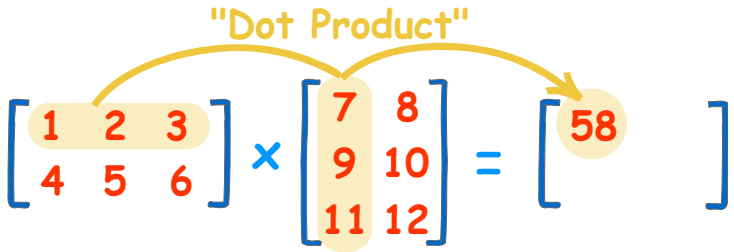
Write a python script where

- ★Write a function to multiply a matrix nxm for a scalar number.
- ★Declare the matrix of the previous example as a list of lists
- ★Declare a scalar number
- ★Multiply the matrix for the scalar
- ★Print the result

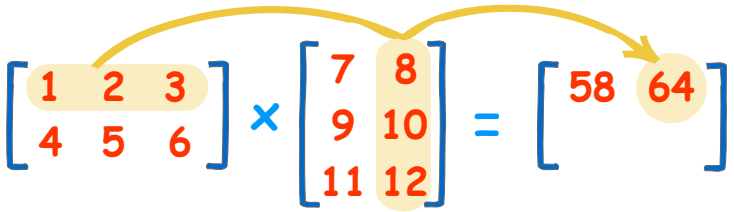
Exercise 1: matrix x scalar

```
a=3
b=[[3,6,9],
   [1,2,3],
   [2,4,8]]

def matrix_per_scalar(matrix, scalar):
    result=[]
    for i in range(len(matrix)):
        tmp=[]
        for j in range(len(matrix[i])):
            tmp.append(matrix[i][j]*scalar)
        result.append(tmp)
    return result

def print_matrix(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            print(str((matrix[i][j]))+"\t", end='')
        print("\n")

print("Input:")
print("Scalar=" + str(a))
print("Matrix=")
print_matrix(b)

print("Matrix x scalar multiplication result:")print_matrix(matrix_per_scalar(b,a))
```

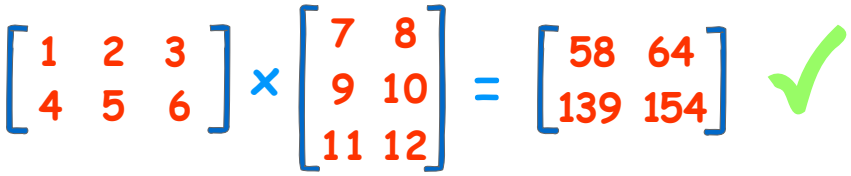
Multiply matrices: Multiplying a Matrix by Another Matrix



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



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



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

Exercise 2: matrix x matrix

Write a python script where

- ★Write a function to multiply a matrix nxm for a matrix mxn
- ★Write a function to print such kind of matrix
- ★Declare the two matrices as list of lists
- ★Multiply the two matrices
- ★Print the result

Exercise 2: matrix x matrix

```
# Program to multiply two matrices
# using nested loops
# 3x3 matrix
A = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
# 3x4 matrix
B = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]
def print_matrix(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            print(str((matrix[i][j]))+"\t", end='')
        print("\n")
def matrix_x_matrix(X, Y):
    # iterate through rows of X
    # result is 3x4
    result = [[0,0,0,0], [0,0,0,0], [0,0,0,0]]
    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]
    return result
```

```
# Main:
print("Input")
print("A = ")
print_matrix(A)
print("B = ")
print_matrix(B)
print("Output AxB")
print_matrix(matrix_x_matrix(A, B))
```

Output:

```
Input
A =
12      7       3
4       5       6
7       8       9

B =
5       8       1       2
6       7       3       0
4       5       9       1

Output AxB
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.

The Anonymous Functions or Lambdas

Anonymous functions or lambdas are small functions which do not need a name (i.e., an identifier).
In Python an anonymous function has 3 parts:

- The lambda keyword, used in place of the keyword ‘def’ used for generic functions
- A set of parameters (can take any number of parameters)
- The function body, which can contain only one expression (in one line of code).

Syntax:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Features:

- The lambda function return just one value in the form of an expression.
- The lambda function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Input parameters

A script can require one or more input parameters.

There are different ways to provide input parameters to a script:

- by command line
- by user
- by an input file

Input parameters by command line.sys.argv

A script requiring parameters can be executed with:

```
$ python script.py param_1 param_2 param_3 …… param_n
```

- The argv[*] provided by tye sys module can be used to read the input parameters:
  - argv[0]: contains the script name
  - argv[1]: param_1
  - ……
  - argv[i]: param_i

Example: command line input (try)

```
# script reqiring 2 input parameters
import sys

usage="""Requires two parameters (param1, param2)
Usage: python script.py param1 param2"""

if len(sys.argv) < 3:
    print('The script: ',sys.argv[0],usage)
    sys.exit(0) # exits after help printing

# read the two input parameters
param1 = sys.argv[1]
param2 = sys.argv[2]

# output the read parameters
print('''The two parameters  received as input
for the script are:\n ''',param1, param2)
```

Input parameters user provided

The input parameters provided by the user can be read from the standard input (stdin) using the function input()
Example (try):

```
# the script takes from the user two input parameters
import sys

while(True):
    print('PLEASE INSERT AN INTEGER NUMBER IN THE RANGE 0-10')
    param1 = input()
    if int(param1) in range(11):
        while(True):
            print( 'PLEASE INSERT A CHAR PARAMETER IN [A,B,C]')
            param2 = input()
            if param2 in ['A','B','C']:
                print('uso I due parametri passati dall utente: ',param1,param2)
                sys.exit()
            else: print('TRY AGAIN PLEASE')
    else: print('TRY AGAIN PLEASE')
```

Input parameters from file

```
infile='mydata.dat'
outfile='myout.dat'

indata = open( infile, 'r')
linee=indata.readlines()
indata.close()
processati=[ ]
x=[ ]
for el in linee:
    valori = el.split()
    x.append(float(valori[0])); y = float(valori[1])
    processati.append(f(y))

outdata = open(outfile, 'w')
i=0
for el in processati:
    outdata.write('%g %12.5e\n' % (x[i],el))
    i+=1
outdata.close()
```

```
def f(y):
   if y >= 0.0:
      return y**5*math.exp(-y)
   else:
      return 0.0
```

```
cat mydata.dat
2     16
13     5
19.3 11
```

Format output: https://www.python-course.eu/python3_formatted_output.php

Input parameters from file

You can read the file with file.read()

```
file = open('.env', "r")
filecontent = file.read()
print("File content:")
print(filecontent)
my_line = ""

for line in filecontent.splitlines():
    print("Working on line", line)
    if line.find("DB_DATABASE="):
        print("Found line containing DB_DATABASE=")
        break
```

Source file:

```
cat .env
DB_HOST= http://localhost/
DB_DATABASE= bheng-local
DB_USERNAME= root
DB_PASSWORD= 1234567890
UNIX_SOCKET= /tmp/mysql.sock
```

Next lesson will go deeply on structured data and how to red them from files

The Anonymous Functions or Lambdas

Anonymous functions or lambdas are small functions which do not need a name (i.e., an identifier).
In Python an anonymous function has 3 parts:

- The lambda keyword, used in place of the keyword ‘def’ used for generic functions
- A set of parameters (can take any number of parameters)
- The function body, which can contain only one expression (in one line of code).

Syntax:

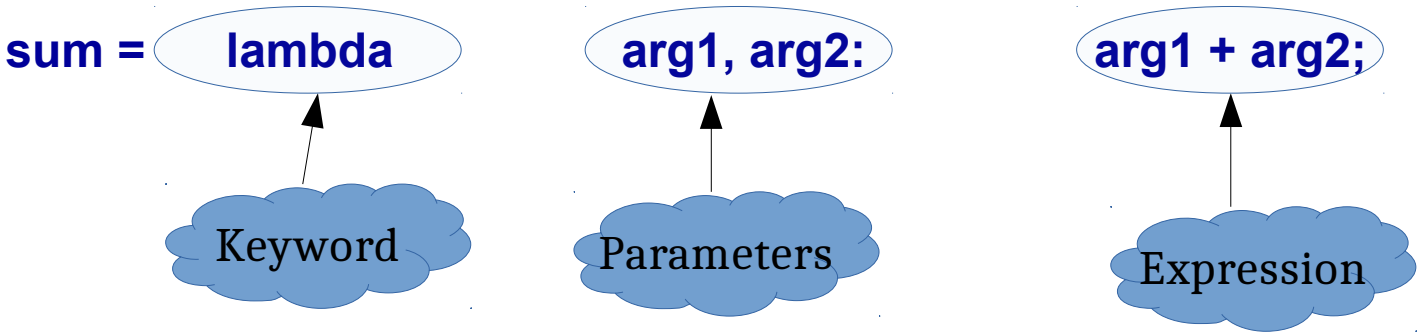
```
lambda [arg1 [,arg2,.....argn]]:expression
```

Features:

- The lambda function return just one value in the form of an expression.
- The lambda function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Example 1: The Lambda/Anonymous Functions

```
#!/usr/bin/python

# Function definition is here
```




```
# Now you can call sum as a function
print("Value of total : ", sum( 10, 20 ))
print( "Value of total : ", sum( 20, 20 ))
```

When the above code is executed, it produces the following result:

```
Value of total :  30
Value of total :  40
```

Example 2: The print and lambda function (1)

Code:

```
string='Hello World!'
print(lambda string : print(string))
```

Works with python3 where print is a function (and a function application is an expression, so it will work in a lambda). In python2 print is a statement and this example does not work.

Output:

```
$ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> string='Hello World!'
>>> print(lambda string : print(string))
<function <lambda> at 0x7fe0922ebd90>
```

Explanation:

Define a string

Declare a lambda that calls a print statement

prints the result, passing the string as parameter.

Why doesn't the program print the string we pass?

Because the lambda itself returns a function object.

The external print instruction prints the result of the lambda function, i.e. the function object and the memory location where it is stored.