Python Lecture 1 – Introduction

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

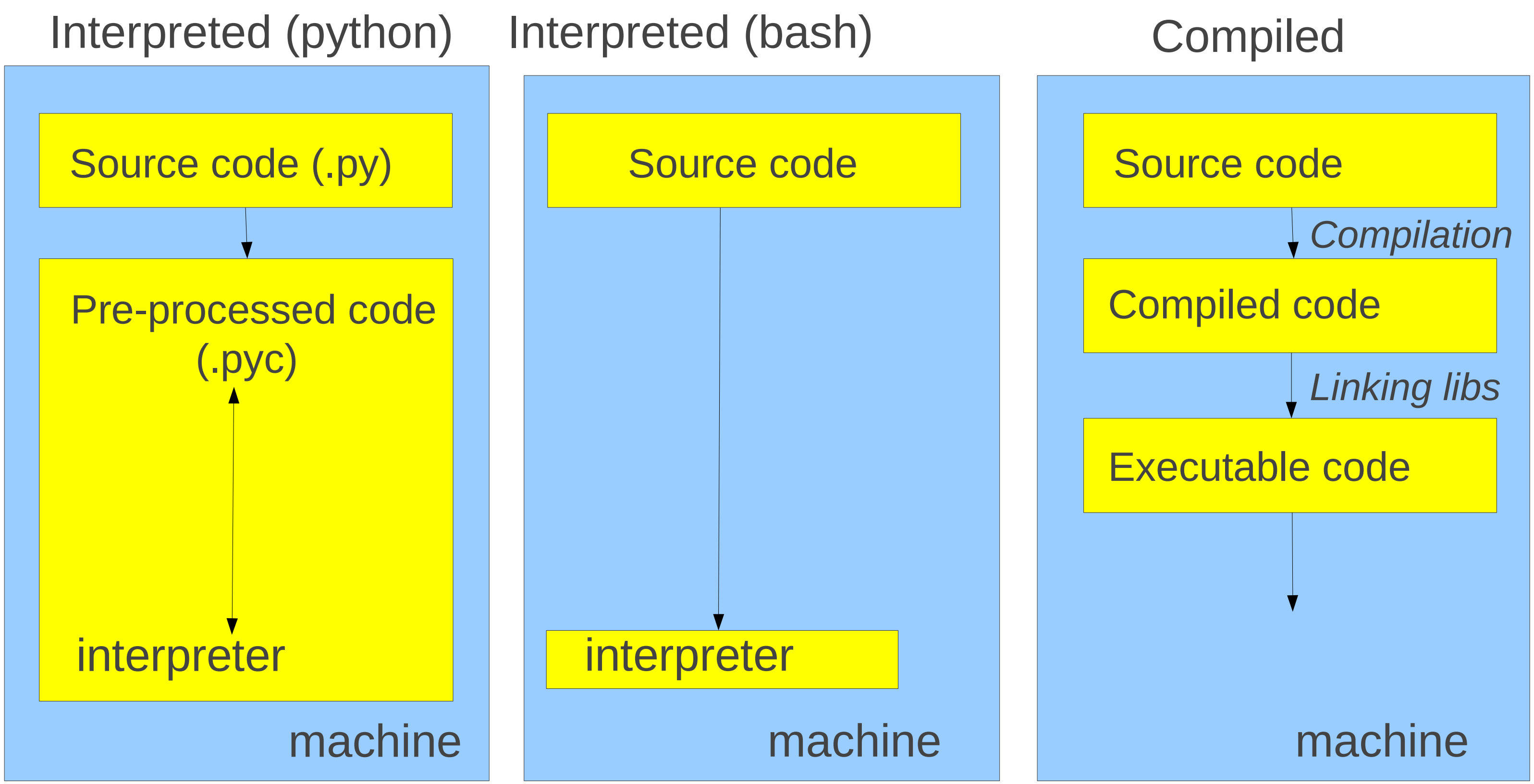
https://github.com/bertocco/bash_lectures

Compiled vs. interpreted languages

Programming languages generally are split in two categories: Compiled or Interpreted.

Compiled	Interpreted
C/C++	Python
Fortran	Perl

Compiled vs. interpreted languages



Interpreted languages:
old-style interpreted languages (like bash): the code is saved in the same format that you entered
new-style interpreted languages (like python): the code is pre-processed to produce a bytecode (similar to machine language) and then executed by the interpreter (virtual machine).
Compiled languages: the code is reduced to a set of machine-specific instructions before being saved as an executable file. The executable file is executed directly on the machine

Compiled vs. interpreted languages

★ Execution speed: compiled languages generally run faster than interpreted ones because interpreted programs must be reduced to machine instructions at runtime. Morover the compilation can be optimized

★ Development speed: It is usually easier to develop applications in an interpreted environment because write code is easier (languages are more high level); errors can be fixed as soon as the interpreter detect them without re-compile.

★ Code portability: means run on hardware/software platforms different from which used to develop the code.

- Compiled code:
  - Is portable on platforms with hardware and softwate similar to which one used for code development and compilation
  - Is portable is only if re-compiled on target platform (need of compiler and lybraries)
- Interpreted code:
  - Is portable if the interpreter is available on the target platform

The python interpreter

★ Python is an interpreted language.

The python interpreter can be used:

- Interactively to interpret a single command or little sets of commands
- Interactively to interpret set of commands collected in a file *.py

  In this case, the interpreter produces files (*.pyc), as intermediate product, and interpret row by row the commantdpresent in the file.

All similarly to bash except for the bytecode production

To fire up the Python interpreter, open up your terminal/console application, and type python or python3 (depending on your installation).

You should see something like this:

```
[bertocco@firiel ~]$ python
Python 2.7.12 |Continuum Analytics, Inc.| (default, Jul  2 2016, 17:42:40)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

★ To get out of the interpreter you can type:

```
>>> quit()
```

Define a variable

A Python variable is a reserved memory location to store values.
The variable must be defined assigning it a value:

```
>>> a=3   #works
>>> b = 3 #works
>>> a
3
>>> c     # does not work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

Rules for Python variable names: A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

- a variable name must start with a letter or the underscore character
- a variable name cannot start with a number
- a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- white spaces and signs with special meanings, as "+" and "-" are not allowed.
- Variable names are case-sensitive (<u>age, Age and AGE are three different variables</u>)

Examples on variable names

- a variable name must start with a letter or the underscore character

```
>>> alfa = 1
>>> alfa
1
>>> beta =10
```

```
>>> _pippo = 1
>>> _pippo
1
```

- a variable name cannot start with a number

```
>>> 1cane
  File "<stdin>", line 1
    1cane
        ^
SyntaxError: invalid syntax
```

- a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

```
>>> urca! = 10
  File "<stdin>", line 1
    urca! = 10
        ^
SyntaxError: invalid syntax
```

- white spaces and signs with special meanings, as "+" and "-" are not allowed

```
>>> a-b=0
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

```
>>> a$b='sara'
  File "<stdin>", line 1
    a$b='sara'
     ^
SyntaxError: invalid syntax
```

Variable types

Each variable in python has a type.

The variable type is not pre-defined, it is resolved at run-time.

In C programs variable declaration is:

int a = 10

float b = 3.4

string str = “pippo”

In python variable declaration is:

a = 10

b = 3.4

str =”pippo”

In python you can do (referring to the previous example):

a = b   # because type is dynamically resolved, i.e. at run-time

In python you can not do (referring to the previous example):

new_val = a + str   # because each variable has a type

                              # (the language is strongly typed)

List of some different variable types

```
x = 123                 # integer
x = 123L                # long integer
x = 3.14                # double float
x = "hello"             # string
x = [0,1,2]             # list
x = (0,1,2)             # tuple
x = open('hello.py', 'r')    # file
x = {1: 'apple', 2: 'ball'}  # dictionary
```

You can also assign a single value to several variables simultaneously multiple assignments.

Variable a,b and c are assigned to the same memory location,with the value of 1

```
a = b = c = 1
```

Example: dynamically resolved types

```
>>> a = 10
>>> b = 3.4
```

It is possible the assignment:

```
>>> a = b
>>> a
3.4
```

```
>>> a = 3
>>> b = 3.4
```

It is possible the sum:

```
>>> a+b
6.4
```

Example: strongly typed variables

```
>>> a=3                # a is resolved as an integer
>>> str = 'mah'        # str is resolved as a string

>>> a + str
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

You can not sum a string with an integer.

Python is dynamically and strongly typed

Python is dynamic typed because variable values are checked during execution.

Python is strongly typed because “at runtime” it doesn’t allow <u>implicit</u> conversions.

Type casting

Casting is the operation to convert a variable value from one type to another.

In Python, the casting must be done explicitly with functions such as int() or float() or str().

Example:

```
>>> x = '100'
>>> y = '-90'

>>> print x + y
100-90

>>> print int(x) + int(y)
10
```

That was the int() function. There's also another very common one which is float() which does basically the same thing:

```
>>> print float(x) + float(y)
10.0
```

Available type casting

int(x [,base]) Converts x to an integer. base specifies the base if x is a string.

long(x [,base]) Converts x to a long integer. base specifies the base if x is a string.

float(x) Converts x to a floating-point number.

complex(real [,imag]) Creates a complex number.

str(x) Converts object x to a string representation.

repr(x) Converts object x to an expression string.

eval(str) Evaluates a string and returns an object.

tuple(s) Converts s to a tuple.

list(s) Converts s to a list.

set(s) Converts s to a set.

dict(d) Creates a dictionary. d must be a sequence of (key,value) tuples.

frozenset(s) Converts s to a frozen set.

chr(x) Converts an integer to a character.

unichr(x) Converts an integer to a Unicode character.

ord(x) Converts a single character to its integer value.

hex(x) Converts an integer to a hexadecimal string.

oct(x) Converts an integer to an octal string.

Be careful in type casting

Type casting is a bit tricky operation.

Example: the string '100.0' can be converted in a float, but not in an integer

```
>>> x = '100.0'
>>> print float(x)
100.0
>>> print int(float(x))
100
>>> print int(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 100.0
```

How to comment code

There are two ways to write comments in python code:

- Single line comments

  # this is a single line comment

  a=13   # this is a single line comment also

- More line comments

  ’’’ You can write a multiple line comment using

  three single quotes’’’

  ””” This is another way to write a multiple line comment

  using three double quotes”””

First python scripts

script_sum.py

```
#!/usr/bin/python
one = 1
two = 2
three = one + two
print(three)
```

Launch the script (first way):

Add execution permissions:

chmod +x script_sum.py

Execute:

./script_sum.py

script_hello.py

```
hello = "hello"
world = "world"
helloworld = hello + " " + world
print(helloworld)
```

Launch the script (second way):

Use the interpreter for execution:

python script_hello.py

Note: anv var PATH or PYTHONPATH must contain ‘python’ command location

How to launch a python script

There are different ways to run a script:

1) ./script.py
   Requires the script pat is in PATH or PYTHONPATH environment variables
2) ./script.py
   Requires the script has execution permissions (chmod +x script.py)
3) From ipython shell

```
$ ipython
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: run hello.py
Hello world
```

Indentation and blocks of code

In python blocks of code (set of instructions to be run as a block, like functions) are denoted by line indentation not by curly braces (as in C or Java, for example). The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Example (indentation.py):

```
x = 2.3
y = 1.2
# test is a function testing if two numbers are equal or one greater then the oter
def test(x,y):
    if x==y:
        print 'the two number are equals'
    elif x > y:
        print ' the first number is the greater'
    else:
        print ' the last number is the greater'

print  'now testing : ', x, y
test(x,y)
for I in range(2,5,1):
    for J in range(5,1,-1):
        print 'now testing : ', I,J
        test(I,J)
```

Exercise: indentation and blocks of code

Try the two examples in the previous slide writing them on a file and running the file.

Functions: defining a function

A function is a block of code which only runs when it is called and can be run repetitively.

Defining a Function:

- Function blocks begin with the keyword def followed by the function name and parentheses ( ) .
- Any input parameter or argument should be placed within these parentheses.
- The first statement of a function can be an optional statement (the documentation string of the function or docstring)
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Functions: calling a function

Defining a function you specify:
name, parameters and the structure of the block of code.

Given the basic structure of the function, it can executed by calling it from another function, from a python script or directly from the python prompt.

Example:

```
#!/usr/bin/python

# Function definition
def printstring( str ):
   "This prints a passed string into this function"
   print str
   return;

# Function call
printstring("I'm first call to user defined function!")
printstring("Again second call to the same function")
```

Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments:

  the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

- Keyword arguments:

  are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters

- Default arguments:

  are arguments that assume a default value if a value is not provided in the function call for these arguments

- Variable-length arguments:

  when a function has to be processed for more arguments than the specified in the function, the variable-length arguments are used. They are not named in the function definition, unlike required, and default arguments (next lesson).

Example: Required Arguments

```
#!/usr/bin/python

# Function definition is here
def myprint( str ):
   "This function prints the passed string"
   print str
   return;

# Now you can call printme function
myprint()
```

When the code is executed, the following is the result:

```
Traceback (most recent call last):
   File "myprint.py", line 11, in <module>
      myprint();
TypeError: myprint() takes exactly 1 argument (0 given)
```

Example: Keyword Arguments

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints the info passed as parameter"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=30, name="Silvia" )
```

Example: Default Arguments

```
#!/usr/bin/python

# Function definition is here
def printuserinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printuserinfo( age=30, name="Silvia" )
printuserinfo( name="Silvia" )
```

When the script is executed, the result is:

```
Name:  Silvia
Age  30
Name:  Silvia
Age  35
```

Scope of variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

- Global variables
- Local variables

Global vs. local variables

Variables that are defined inside a function body have a local scope,
those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared,
whereas global variables can be accessed throughout the program body by all functions.
When you call a function, the variables declared inside it are brought into scope.

Example (try):

```
#!/usr/bin/python

total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

Output calling the script:

Inside the function local total : 30
Outside the function global total : 0

Pass function arguments by reference vs. value

All parameters (arguments) of functions in Python are passed by reference:
if you change what a parameter refers to within a function,
then the change also reflects back in the calling function.

Example:

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function:\n ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: \n", mylist
```

Output:

```
Values inside the function:
[10, 20, 30, [1, 2, 3, 4]]

Values outside the function:
[10, 20, 30, [1, 2, 3, 4]]
```

Be careful duplicating variable names

```
!/usr/bin/python
# Function definition is here
def changelist( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4];      # This assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Function call
mylist = [10,20,30];
changelist( mylist );
print "Values outside the function: ", mylist
```

The parameter mylist is local to the function changelist. Changing mylist within the function does not affect mylist outside the function.

Output:

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

The Anonymous Functions

Anonymous functions are not declared in the standard manner by using the def keyword. The lambda keyword is used to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- They cannot contain commands or multiple expressions.
- An anonymous 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.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of lambda functions contains only a single statement, which is as follows:

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

Example: The Anonymous Functions

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# 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
```

return statement

The statement return [expression] exits a function,
optionally passing back an expression to the caller.
A return statement with no arguments is the same as return None.

Example of return value

```
#!/usr/bin/python
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

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

```
Inside the function :  30
Outside the function :  30
```

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) < 2:
    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): # notare che raw_input restituisce una stringa
        while(True):
            print 'PLEASE INSERT A CHAR PARAMETER IN [A,B,C]'
            param2 = raw_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'
```

functions input() and raw_input()

Python 2:
raw_input() takes exactly what the user typed and passes it back as a string.

input() first takes the raw_input() and then performs an eval() on it as well.

The main difference is that input() expects a syntactically correct python statement where raw_input() does not.

Python 3:
raw_input() was renamed to input() so now input() returns the exact string.
old input() was removed.

If you want to use the old input(), meaning you need to evaluate a user input as a python statement, you have to do it manually by using eval(input()).

Example input() and raw_input() in python 2

```
In [17]: raw_input("Whats your name?")
Whats your name?pippo
Out[17]: 'pippo'

In [18]: input("Whats your name?")
Whats your name?pluto
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-f3bdb23bc89a> in <module>()
----> 1 input("Whats your name?")

<string> in <module>()

NameError: name 'pluto' is not defined

In [19]: input("Whats your name?")     # I have to provide a row
Whats your name?'pluto'
Out[19]: 'pluto'
```

eval()

The eval() method parses the expression passed to this method and runs python expression (code) within the program.

The eval() method runs the python code (which is passed as an argument) within the program.

Try

```
>>> help(eval)
```

to better understand the command.

Example:

```
x = 1
print(eval('x + 1'))
```

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

Modules

Modules are file containing Python statements and definitions.

A file containing python code is called a module
If the file is *my_lib.py*, the module name is *my_lib*

Modules are useful to break down large programs into small manageable and organized files.

Modules provide re-usability of code: useful functions can be put in a module and later imported in another module/script and re-used.

How to import module:
import my_module

How to import module by name:
import my_module as example

How to import a single function by a module:
from my_module import my_function

How to import all names in a module:
*from my_module import **

Example: module imports

File (module) my_libs.py:

```
def add(a, b):
  #This program adds two numbers and return the result
  result = a + b
  return result
def multiply(a,b)
  #This program multiply two numbers and return the result
  result = a * b
  return result
```

- Import the entire module (my_modules_example_1.py)

```
import my_libs
print “add 4 and 5.5. Result:”, my_libs.add(4,5.5)
```

- Import a single function (my_modules_example_2.py)

```
from my_libs import multiply
print "Multiply 4 X 5. Result:", multiply(4, 5)
```

- Import the entire module by name(my_modules_example_3.py)

```
import my_libs as example
print “add 4 and 5.5. Result:”, example.add(4,5.5)
```

Example: module os (manage OS dialog operations)

```
import os

#namespace of module os
>>> os.curdir
'.'
>>> os.getenv('HOME')
'/home/bertocco'
>>> os.listdir('.')
['my_modules_example_1.py',
 'read_by_line.py',
 '.mozilla',
 '.bash_logout',
…..
]
In [2]: import os

In [3]: os.defpath
Out[3]: ':/bin:/usr/bin'
```

Module Search Path

To import a module, Python looks at several places.
Interpreter first looks for a built-in module then the search is in this order:

- The current directory
- PYTHONPATH (an environment variable with a list of directory)
- The installation-dependent default directory

How to reload a Module

The Python interpreter imports a module only once during a session.
This makes things more efficient.

If the module is changed during the course of the program, we would have to reload it. To reload the module you have two ways:

- Restart the interpreter (not much clean).
- Use the function reload() inside the imp module. Example:

In [29]: import my_libs

In [30]: import imp

In [31]: imp.reload(my_libs)
Out[31]: <module 'my_libs' from 'my_libs.pyc'>

Introspection

Introspection of a language is the ability of the language itself to provide information of its objects at runtime.

Python ha a very good support for introspection.

The interpreter can be used interactively to better know and understand the code.

Following the description of useful python built-in functions for introspection

dir()

The dir() function can be used to find out names that are defined inside a module.

For example, we have defined the functions add(a,b) and multiply(a,b) in the module my_libs. We can find them:

```
In [32]: dir(my_libs)
Out[32]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 'add',
 'multiply']
```

names that begin with an underscore are default Python attributes associated with the module (we did not define them ourself).

All the names defined in the current namespace can be found out using the dir() function without any arguments. Example (try):

```
dir()
```

help()

The function help is available for each module/object and allows to know the documentation for each function.

Try (in the interpreter) the commands:

import math

dir()

help(math.acos)

Example:

```
In [8]: import math
In [9]: dir(math)
Out[9]:
['__doc__',
 '__name__',
 '__package__',
 'acos',
 'acosh',
……….
In [19]: help(math.acos)
Help on built-in function acos in module math:
acos(...)
    acos(x)
    Return the arc cosine (measured in radians) of x.
```

type()

The type function allows to

```
In [1]: a=5
In [2]: type a
  File "<ipython-input-2-e92615c5fd0c>", line 1
    type a
         ^
SyntaxError: invalid syntax
In [3]: type(a)
Out[3]: int



In [5]: l = [1, "alfa", 0.9, (1, 2, 3)]
In [6]: print [type(i) for i in l]
[<type 'int'>, <type 'str'>, <type 'float'>, <type 'tuple'>]
```

pydoc

Pydoc is a python tool for introspection.
It provides information enclosed in a module in a clear and compact manner.

Pydoc uses the doc string __doc__ and other standard attributes of objects (__name__, __file__, ...).

```
$ pydoc os
Help on module os:
NAME
os - OS routines for Mac, DOS, NT, or Posix depending on what system we're on.
FILE
/usr/lib64/python2.4/os.py
DESCRIPTION
This exports:
- all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.
- os.path is one of the modules posixpath, ntpath, or macpath
- os.name is 'posix', 'nt', 'os2', 'mac', 'ce' or 'riscos'
- os.curdir is a string representing the current directory ('.' or ':')
……………………...
```

The zen of python

Type in the interpreter: import this

Output: the zen of python

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Exercise

Go to page
https://docs.python.org/2/tutorial/introduction.html
to the paragraph
“Using Python as a Calculator”
practice with the described operators.
Just to practice with the language,
write a module containing a python function executing the operation for each operator,
write your own script importing the module, reading input from command line,
executing the operations using the functions previously coded,
print the operators and the result for each function