Programming in Python

GAVRILUT DRAGOS

COURSE 1

Administrative

Final grade for the Python course is computed using Gauss over the total points accumulated.

One can accumulate a maximum of 100 of points:

- Maximum 70 points from the laboratory examination
- Maximum 30 points at the final examination (course)

The laboratory examination consists in 2 test:

- First test → 30 points
- Second test → 40 points

The minimum number of points that one needs to pass this exam:

- Minimum 25 points accumulated from the first and the seconds laboratory tests
- Minimum 10 points from the final examination (course)

Course page: https://sites.google.com/site/fiipythonprogramming/home

History

1980 – first design of Python language by Guido van Rossum

1989 – implementation of Python language started

2000 – Python 2.0 (garbage collector, Unicode support, etc)

2008 – Python 3.0

Current Versions:

- 2.x → 2.7.14 (available from 16.Sep.2017)
- 3.x → 3.6.3 (available from 3.Oct.2017)

Download python from: https://www.python.org

Help available at : https://docs.python.org/2.7/ and https://docs.python.org/3.6/

Python coding style: https://www.python.org/dev/peps/pep-0008/#id32

General information

Companies that are using Python: Google, Reddit, Yahoo, NASA, Red Hat, Nokia, IBM, etc

TIOBE Index for September 2017 → **Python** is ranked no. 5

Programming Language Index PyPL ranks **Python** as no.2

Github ranks **Python** as no. 3

Default for Linux and Mac OSX distribution (both 2.x and 3.x versions)

Open source

Support for almost everything: web development, mathematical complex computations, graphical interfaces, etc.

.Net implementation → IronPython (http://ironpython.net)

Characteristics

- Un-named type variable
- Duck typing → type constrains are not checked during compilation phase
- Anonymous functions (lambda expressions)
- Design for readability (white-space indentation)
- Object-oriented programming support
- Reflection
- Metaprogramming → the ability to modify itself and create new types during execution

Zen Of Python

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.

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.

Unless explicitly silenced.

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!

Python editors

Notepad++ ➔ https://notepad-plus-plus.org/download/v7.5.1.html

Komodo IDE ➔ http://komodoide.com

PyCharm ➔ https://www.jetbrains.com/pycharm/

VSCode ➔ https://marketplace.visualstudio.com/items?itemName=donjayamanne.python

Eclipse ➔ http://www.liclipse.com

PyDev ➔ https://wiki.python.org/moin/PyDev

WingWare ➔ http://wingware.com

PyZO ➔ http://www.pyzo.org

Thonny ➔ http://thonny.cs.ut.ee

……..

First Python program

The famous “Hello world”

C/C++

```
void main(void)
{
        printf("Hello world");
}
```

Python 2.x

```
print "Hello world"
```

Python 3.x

```
print ("Hello world")
```

Variables

Variable are defined and use as you need them.

Python 2.x / 3.x

```
x = 10                  #x is a number
s = "a string"          #s is a string
b = True                #b is a Boolean value
```

Variables don't have a fixed type – during the execution of a program, a variable can have multiple types.

Python 2.x / 3.x

```
x = 10
#do some operations with x
x = "a string"
#x is now a string
```

Basic types

Python 2.x / 3.x

```
x = 10  #x is an integer (32 bit precision)
x = 99999999999999999999999999999999 #x is a long (unlimited precision)
x = 1.123 #x is an float
x = 1.2j #x is a complex number
x = True #x is a bool ( a special case of integer )
x = None #x is a NoneType(the closest C/C++ equivalent is NULL/nullptr)
```

Python 2.x

```
x = 10
print x, type (x)
```

Output

```
10 <type 'int'>
```

Python 3.x

```
x = 10
print (x, type (x))
```

Output

```
10 <class 'int'>
```

Numerical operations

Arithmetic operators (+, - , *, /, %) – similar to C like languages

Python 2.x / 3.x

```
x = 10+20*3          #x will be an integer with value 70
x = 10+20*3.0        #x will be an float with value 70.0
```

Operator ** is equivalent with the pow function from C like languages

Python 2.x / 3.x

```
x = 2**8             #x will be an integer with value 256
x = 2**8.1           #x will be an float with value 274.374
```

A number can be casted to a specific type using **int** or **float** method

Python 2.x / 3.x

```
x = int(10.123)      #x will be an integer with value 10
x = float(10)        #x will be an float with value 10.0
```

Numerical operations

Division operator has a different behavior in Python 2.x and Python 3.x

Python 2.x / 3.x

```
x = 10.0/3          #x will be a float with value 3.3333
x = 10.0%3          #x will be a float with value 1.0
```

Division between integers is interpreted differently

Python 2.x

```
x = 10/3
#x is 3 (int)
```

Python 3.x

```
x = 10/3
#x is 3.33333 (float)
```

A special operator exists // that means integer division (for integer operators)

Python 2.x / 3.x

```
x = 10.0//3         #x will be a float with value 3.0
x = 11.9//3         #x will be a float with value 3.0
```

Numerical operations

Bit-wise operators (& , | , ^ , <<, >>). In particular & operator can be use to make sure that a behavior specific to a C/C++ operation can be achieve

C/C++

```
void main(void)
{
        unsigned int x;
        x = 0xFFFFFFFE;
        x = x + x;
        unsigned char y;
        y = 123;
        y = y + y;
}
```

Python 2.x / 3.x

```
x = 0xFFFFFFFE
x = (x + x) & 0xFFFFFFFF
y = 123
y = (y + y) & 0xFF
```

Numerical operations

Compare operators (>, < , >=, <=, ==, !=). C/C++ like operators && and || are replaced with **and** and **OR.** Similary ! opeartor is replaced with **not** keyword. However, unlike C/C++ languages Python supports a more mathematical like evaluation.

Python 2.x / 3.x

```
x = 10 < 20 > 15          #x is True
                          #identical to (10<20) and (20>15)
```

Operator != has a form of alias in Python 2.x (similar to Pascal language ➔ <>). For == operator there is also a special keyword "**is**" that can be used. Similar, "**is not**" can be used to describe the != operator.

- In Python 2.x an expression like "x <> y" is equivalent cu "x != y".

All of this operators produce a bool result. There are two special value defined in Python:

- True
- False

String Types

Python 2.x / 3.x

```
s = "a string\nwith lines"
s = 'a string\nwith lines'
s = r"a string\nwithout any line"
s = r'a string\nwithout any line'
```

Python 2.x / 3.x

```
s = """multi-line
string
"""
```

Python 2.x / 3.x

```
s = '''multi-line
string
'''
```

String Types

Strings in python have support for different types of formatting – much like in C/C++ language.

Python 2.x/3.x

```
s = "Name: %8s Grade: %d"%("Ion",10)
```

If only one parameter has to be replaced, the same expression can be written in a simplified form:

Python 2.x/3.x

```
s = "Grade: %d"%10
```

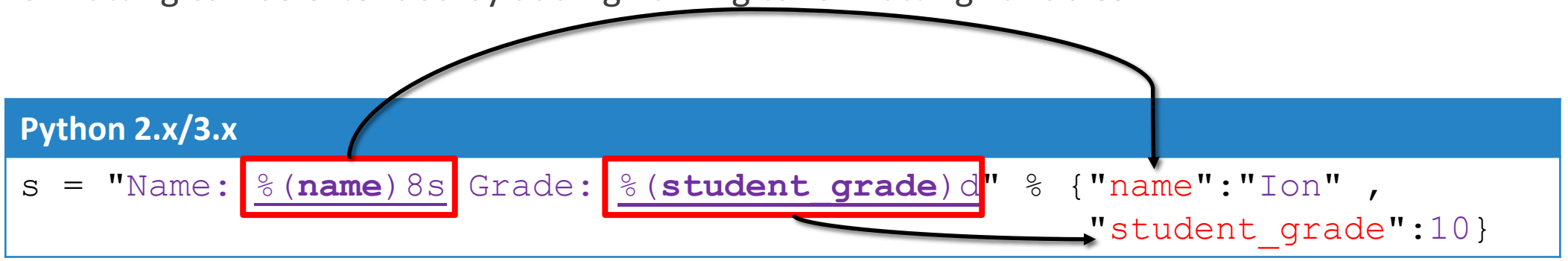
Two special keywords **str** and **repr** can be use to convert variables from any type to string.

Python 2.x/3.x

```
s = str (10)                #s is ”10”
s = repr (10.25)            #s is ”10.25”
```

String Types

Formatting can be extended by adding naming to formatting variables.



Python 2.x/3.x

```
s = "Name: %(name)8s Grade: %(student_grade)d" % {"name":"Ion" ,
                                                  "student_grade":10}
```

A special character "\" can be place at the end of the string to concatenate it with another one from the next line.

Python 2.x/3.x

```
s = "Python"\
"Exam"
#s is "PythonExam"
```

String Types

Strings also support different ways to access characters or substrings

Python 2.x / 3.x

```
s = "PythonExam"     #s is "PythonExam"

s[1]                 #Result is "y" (second character, first index is 0)
s[-1]                #Result is "m" → "PythonExam"(last character)
s[-2]                #Result is "a" → "PythonExam"
s[:3]                #Result is "Pyt" → "PythonExam"(first 3 characters)
s[4:]                #Result is "onExam" → "PythonExam"
                     #(all the characters starting from the 4th character
                     #of the string until the end of the string)
s[3:5]               #Result is "ho" → "PythonExam" (a substring that
                     #starts from the 3rd character until the 5th one)
s[2:-4]              #Result is "thon" → "PythonExam"
```

String Types

Strings also support a variety of operators

Python 2.x / 3.x

```
s = "Python"+"Exam"#s is “PythonExam”
s = "A"+"12"*3      #s is “A121212” ➔ ”12” is multiplied 3 times
"A" in "Python"     #Result is False (”A” string does not exists in
                    #                 ”Python” string)
"A" not in "ABC"    #Result is False (”A” string exists in ”ABC”)
len (s)             #Result is 10 (10 characters in “PythonExam” string)
```

And slicing:

Python 2.x / 3.x

```
s = "PythonExam"    #s is “PythonExam”
s[1:7:2]            #Result is ”yhn” (Going from index 1, to index 7
                    #with step 2 (1,3,5) ➔ PythonExam
```

String Types

Every string is consider a class and has member functions associated with it. This functions are accessible through "." operator.

- Str.**startswith**("…") ➔ checks if a string starts with another one
- Str.**endswith**("…") ➔ checks if a string ends with another one
- Str.**replace**(toFind,replace,[count]) ➔ returns a string where the substring <*toFind*> is replaced by substring <*replace*>. Count is a optional parameter, if given only the firs <*count*> occurrences are replaced
- Str.**index**(toFind) ➔ returns the index of <*toFind*> in current string
- Str.**rindex**(toFind) ➔ returns the right most index of <*toFind*> in current string
- Other functions: **lower**(), **upper**(), **strip**(), **rstrip**(), **lstrip**(), **format**(), **isalpha**(), **isupper**(), **islower**(), **find**(…), **count**(…), etc

String Types

Strings splitting via .**split** function

Python 2.x / 3.x

```
s = "AB||CD||EF||GH"
s.split("||")[2]    #Result is "EF". Split produces an array of 4
                    #elements AB,CD,EF and GH. The second element is EF
s.split("||")[-1]   #Result is "GH".
s.split("||",1)[0]  #Result is "AB". In this case the second parameter
                    #tells the function to stop after <count> (in this
                    #case 1) splits. Split produces an array of 2
                    #elements AB and CD||EF||GH. The fist element is AB
s.split("||",2)[2]  #Result is "EF||GH". Split produces an array of 3
                    #elements AB, CD and CEF||GH.
```

Strings also support another function .**rsplit** that is similar to .**split** function with the only difference that the splitting starts from the end and not from the beginning.

Built-in functions for strings

Python has several build-in functions design to work characters and strings:

- **chr** (*charCode*) ➔ returns the string formed from one character corresponding to the code *charCode. charCode* is an Unicode code value.
- **ord** (character) ➔ returns the Unicode code corresponding to that specific character
- **hex** (number) ➔ converts a number to a lower-case hex representation
- **oct** (number) ➔ converts a number to a base-8 representation
- **format** ➔ to format a string with different values

Statements

Python is heavily based on indentation to express a complex instruction

C/C++

```
if (a>b)
{
        a = a + b
        b = b + a
}
```

Python 2.x/3.x

```
if a>b:
        a = a + b
        b = b + a
```

Python 2.x/3.x

```
if a>b:
________a = a + b
________b = b + a
```

Complex instruction

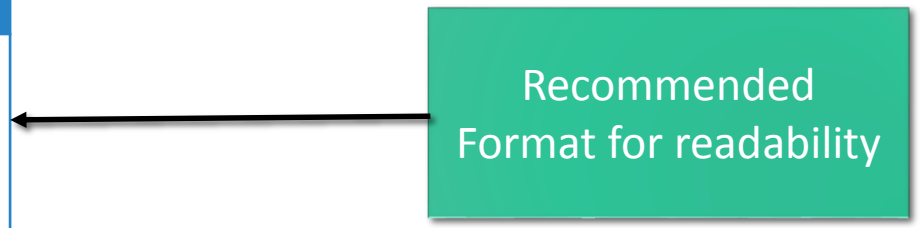
Statements

While python coding style recommends using indentation, complex instruction can be written in a different way as well by using a semicolon and add simple expression on the same line:

For example the following expression:

Python 2.x/3.x

```
if a>b:
        a = a + b
        b = b + a
        b = a * b
```



Can also be written as follows:

Python 2.x/3.x

```
if a>b: a = a + b ; b = b + a ; b = a * b
```

IF-Statement

Python 2.x/3.x

```
if expression:
        complex or simple statement
```

Python 2.x/3.x

```
if expression:
        complex or simple statement
else:
        complex or simple statement
```

Python 2.x/3.x

```
if expression:
        complex or simple statement
elif expression:
        complex or simple statement
```

Python 2.x/3.x

```
if expression:
        complex or simple statement
elif expression:
        complex or simple statement
elif expression:
        complex or simple statement
elif expression:
        complex or simple statement
…
else:
        complex or simple statement
```

SWITCH/CASE - Statements

Python **does not have** a special keyword to express a switch statement. However, if-elif-else keywords can be used to describe the same behavior.

C/C++

```
switch (var) {
        case value_1:
                statements;
                break;
        case value_2:
                statements;
                break;
...
        default:
                statements;
                break;
}
```

Python 2.x/3.x

```
if var == value_1:
        complex or simple statement
elif var == value_2:
        complex or simple statement
elif var == value_3:
        complex or simple statement
…
else:   #default branch from switch
        complex or simple statement
```

WHILE - Statement

C/C++

```
while (expression) {
        statements;
}
```

Python 2.x/3.x

```
while expression:
        complex or simple statement
```

Python 2.x/3.x

```
while expression:
        complex or simple statement
else:
        complex or simple statement
```

Python 2.x/3.x

```
a = 3
while a > 0:
        a = a - 1
        print (a)
else:
        print ("Done")
```


Output

```
2
1
0
Done
```

WHILE - Statement

The **break** keyword can be used to exit the while loop. Using the **break** keyword will not move the execution to the **else** statement if present !

Python 2.x/3.x

```
a = 3
while a > 0:
        a = a - 1
        print (a)
        if a==2: break
else:
        print ("Done")
```


Output

```
2
```

WHILE - Statement

Similarly the **continue** keyword can be used to switch the execution from the while loop to the point where the while condition is tested.

Python 2.x/3.x

```
a = 10
while a > 0:
        a = a - 1
        if a % 2 == 0: continue
        print (a)
else:
        print ("Done")
```

Output

```
9
7
5
3
1
Done
```

DO…WHILE - Statement

Python **does not have** a special keyword to express a do … while statement. However, using the **while…else** statement a similar behavior can be achieved.

C/C++

```
do {
        statements;
}
while (test_condition);
```

Python 2.x/3.x

```
while test_condition:
        statements
else:
        statements
```

Exemple:

C/C++

```
do {
        x = x - 1;
}
while (x > 10);
```

Python 2.x/3.x

```
while x > 10:
        x = x - 1
else:
        x = x - 1
```

FOR- Statement

For statement is different in Python that the one mostly used in C/C++ like languages. It resembles more a foreach statement (in terms that it only iterates through a list of objects, values, etc). Besides this, all of the other known keywords associated with a for (**break** and **continue**) work in a similar way.

Python 2.x/3.x

```
for <list_of_iterators_variables> in <list>:
        complex or simple statement
```

Python 2.x/3.x

```
for <list_of_iterators_variables> in <list>:
        complex or simple statement
else:
        complex or simple statement
```

FOR- Statement

A special keyword **range** that can be use to simulate a C/C++ like behavior.

Python 2.x/3.x

```
for index in range (0,3):
        print (index)
```

Output

```
0
1
2
```

Python 2.x/3.x

```
for index in range (0,3):
        print (index)
else:
        print ("Done")
```

Output

```
0
1
2
Done
```

FOR- Statement

range operator has a different behavior in Python 2.x and Python 3.x. In Python 2.x it returns a list of numbers (range(0,999) → will return a list containing 1000 numbers) while in Python 3.x it returns an iter-eable object that will iterate from 0 to 1000. While most of the time, this is not an issue, when dealing with large intervals it might be problematic. Because of this, Python 2.x contains another keyword (**xrange**) that has the same functionality as the range keyword in Python 3.x

range keyword is declared as follows **range** (*start*, *end*, [*step*])

Python 2.x/3.x

```
for index in range (0,8,3):
        print (index)
```

Output

```
0
3
6
```

for statement will be further discuss in the course no. 2 after the concept of list is presented.

Functions

Functions in Python are defined using **def** keyword

Python 2.x/3.x

```
def function_name (param1,param2,… paramn ):
        complex or simple statement
```

Parameters can have a default value.

Python 2.x/3.x

```
def function_name (param1,param2 [= defaultVal],… paramn[= defaultVal] ):
        complex or simple statement
```

And finally, **return** keyword can be used to return values from a function. There is no notion of void function (similar cu C/C++ language) → however, this behavior can be duplicated by NOT using the **return** keyword.

Functions

Simple example of a function that performs a simple arithmetic operation

Python 2.x/3.x

```
def myFunc (x, y, z):
        return x * 100 + y * 10 + z
print (myFunc (1,2,3) )                                    #Output:123
```

Parameters can be explicitly called

Python 2.x/3.x

```
def sum (x, y, z):
        return x * 100 + y * 10 + z
print ( sum (z=1,y=2,x=3) )                                #Output:321
```

Functions

Function parameters can have default values. Once a parameter is defined using a default value, every parameter that is declared after it should have default values.

Python 2.x/3.x

```
def myFunc (x, y=6, z=7):
        return x * 100 + y * 10 + z
print (myFunc (1) )                          #Output:167
print (myFunc (2,9) )                        #Output:297
print (myFunc (z=5,x=3) )                    #Output:365
print (myFunc (4,z=3) )                      #Output:463
print (myFunc (z=5) )                        #ERROR: missing x
```

Python 2.x/3.x

```
def myFunc (x=2, y, z=7):
        return x * 100 + y * 10 + z
```


Code will not compile as x has a default value, but Y does not !

Functions

A function can return multiple values at once. This will also be discuss in course no. 2 along with the concept of touple.

Python also uses **global** keyword to specify within a function that a specific variable is in fact a global variable.

Python 2.x/3.x

```
x = 10
def ModifyX ():
        x = 100
ModifyX ()
print ( x ) #Output:10
```

Python 2.x/3.x

```
x = 10
def ModifyX ():
        global x
        x = 100
ModifyX ()
print ( x ) #Output:100
```

Functions

Functions can have a variable – length parameter (similar to the … from C/C++).
It is precede by "*" operator.

Python 2.x/3.x

```
def multi_sum (*list_of_numbers):
        s = 0
        for number in list_of_numbers:
                s += number
        return s

print ( multi_sum (1,2,3) )                    #Output:6
print ( multi_sum (1,2) )                      #Output:3
print ( multi_sum (1) )                        #Output:1
print ( multi_sum () )                         #Output:0
```

Functions

Functions can return values of different types. In this case you should check the type before using the return value.

Python 2.x/3.x

```
def myFunction(x):
        if x>0:
                return "Positive"
        elif x<0:
                return "Negative"
        else:
                return 0
result = myFunction (0)
if type(result) is int:
        print("Zero")
else:
        print(result)
```

Functions

Functions can also contain another function embedded into their body. That function can be used to compute results needed in the first function.

Python 2.x/3.x

```
def myFunction(x):
        def add (x,y):
                return x+y
        def sub(x,y):
                return x-y

        return add(x,x+1) + sub(x,2):
print myFunction (5)
```

The previous code will print 14 into the screen.

Functions

Functions can also be recursive (see the following implementation for computing a Fibonacci number)

Python 2.x/3.x

```
def Fibonacci (n):
        if n == 1:
                return 1
        elif n == 2:
                return 1
        else:
                return Fibonacci (n-1) + Fibonacci (n-2)

print ( Fibonacci (10) )
```

The previous code will print 55 into the screen.

Functions

It is recommended to add a short explanation for every defined function by adding a multi-line string immediately after the function definition
https://www.python.org/dev/peps/pep-0257/#id15

Python 2.x/3.x

```
def Fibonacci (n):
        """
        Computes the n-th Fibonaci number using recursive calls
        """
        if n == 1:
                return 1
        elif n == 2:
                return 1
        else:
                return Fibonacci (n-1) + Fibonacci (n-2)
```

How to create a python file

- Create a file with the extension .py
- If you run on a Linux/OSX operation system you can add the following line at the beginning of the file (the first line of the file):
  - `#!/usr/bin/python3` ➔ for python 3
  - `#!/usr/bin/python` ➔ for python (current version – usually 2)
- These lines can be added for windows as well (“#” character means comment in python so they don’t affect the execution of the file too much
- Write the python code into the file
- Execute the file.
  - You can use the python interpreter directly (usually C:\Python27\python.exe or C:\Python36\python.exe for Windows) and pass the file as a parameter
  - Current distributions of python make some associations between .py files and their interpreter. In this cases you should be able to run the file directly without using the python executable.

Programming in Python

GAVRILUT DRAGOS

COURSE 2

Lambda functions

A lambda function is a function without any name. It has multiple roles (for example it is often use as a pointer to function equivalent when dealing with other functions that expect a callback).

Lambdas are useful to implement closures.

A lambda function is defined in the following way:

lambda *<list_of_parameters>* **:** *return_value*

The following example uses lambda to define a simple addition function

Python 2.x / 3.x (without lambda)	Python 2.x / 3.x (with lambda)

`def addition (x,y):`
`  return x+y`
`print (addition (3,5))`
	`addition = lambda x,y: x+y`
	`print (addition(3,5))`

Lambda functions

Lambdas are bind during the run-time. This mean that a lambda with a specific behavior can be build at the run-time using the data dynamically generated.

Python 2.x / 3.x

```
def CreateDivizibleCheckFunction(n):
        return lambda x: x%n==0
fnDiv2 = CreateDivizibleCheckFunction (2)
fnDiv7 = CreateDivizibleCheckFunction (7)
x = 14
print ( x, fnDiv2(x), fnDiv7(x) )
```

In this case fnDiv2 and fnDiv7 are dynamically generated.

This programming paradigm is called **closure**.

Output

```
14 True True
```

Sequences

A sequence in python is a data structure represented by a vector of elements that don't need to be of the same type.

Lists have two representation in python:

- **list** ➔mutable vector (elements from that list can be added, deleted, etc). List can be defined using […] operator or the **list** keyword
- **tuple** ➔ immutable vector (the closest equivalent is a constant list) ➔ addition, deletion, etc operation can not be used on this type of object. A tuple is usually defined using (…) or by using the **tuple** keyword

list and **tuple** keywords can also be used to initialized a tuple or list from another list of tuple

Sequences

Python 2.x / 3.x

```
x = list ()              #x is an empty list
x = []                   #x is an empty list
x = [10,20,"test"]       #x is list
x = [10,]                #x is list containing [10]
x = [1,2] * 5            #x is list containing [1,2,  1,2,  1,2,  1,2,  1,2]
x,y = [1,2]              #x is 1 and y is 2
```

```
x = tuple ()             #x is an empty tuple
x = ()                   #x is an empty tuple
x = (10,20,"test")       #x is a tuple
x = 10,20,"test"         #x is a tuple
x = (10,)                #x is tuple containing (10)
x = (1,2) * 5            #x is tuple containing (1,2,  1,2,  1,2,  1,2,  1,2)
x = 1,2 * 5              #x is tuple containing (1,10)
x,y = (1,2)              #x is 1 and y is 2 (the same happens for x,y = 1,2 )
```

Sequences

Elements from a list can be accessed in the following way

Python 2.x / 3.x

```
x = ['A', 'B', 2, 3, 'C']

x[0]      #Result is A
x[-1]     #Result is C
x[-2]     #Result is 3
x[:3]     #Result is ['A', 'B', 2]
x[3:]     #Result is [3, 'C']
x[1:3]    #Result is ['B', 2]
x[1:-3]   #Result is ['B']
```

Sequences

Elements from a tuple can be accessed in the same way

Python 2.x / 3.x

```
x = ('A', 'B', 2, 3, 'C')

x[0]      #Result is A
x[-1]     #Result is C
x[-2]     #Result is 3
x[:3]     #Result is ('A', 'B', 2)
x[3:]     #Result is (3, 'C')
x[1:3]    #Result is ('B', 2)
x[1:-3]   #Result is ('B')
```

Sequences

tuple and **list** keywords can also be used to convert a tuple to a list and vice-versa.

Python 2.x / 3.x

```
x = ('A', 'B', 2, 3, 'C')
y = list (x) #y = ['A', 'B', 2, 3, 'C']
```

```
x = ['A', 'B', 2, 3, 'C']
y = tuple (x) #y = ('A', 'B', 2, 3, 'C')
```

Both lists and tuples can be concatenated, **but not with each other**.

Python 2.x / 3.x

```
x = ('A', 2)
y = ('B', 3)
z = x + y
#z = ('A', 2, 'B', 3)
```

```
x = ['A', 2]
y = ['B', 3]
z = x + y
#z = ['A', 2, 'B', 3]
```

```
x = ('A', 2)
y = ['B', 3]
z = x + y
#!!! Error !!!
```

Sequences

Tuples are also used to return multiple values from a function.

The following example computes both the sum and product of a sequence of numbers

Python 2.x / 3.x

```
def ComputeSumAndProduct(*list_of_numbers):
        s = 0
        p = 1
        for i in list_of_numbers:
                s += i
                p *= i
        return (s,p)

suma,produs = ComputeSumAndProduct(1,2,3,4,5)
#suma =15, produs = 120
```

Sequences

tuple and **list** can also be organized in matrixes:

Python 2.x / 3.x

```
x = ((1,2,3), (4,5,6))
x = ([1,2,3], (4,5,6)) #matrix sub components don't have to be of the
                       #same type
x = ( ((1,2,3), (4,5,6)), ((7,8), (9,10,11, 12)) )
#a matrix does not have to have the same number of elements on each
#dimension
```

```
#the same rules from tuples apply to lists as well
x = [[1,2,3], [4,5,6]]
x = [[1,2,3], (4,5,6)]
```

Sequences

Both **tuples** and **lists** can be enumerated with a **for** keyword:

Python 2.x / 3.x

```
for i in [1,2,3,4,5]:
        print(i)
```

Python 2.x / 3.x

```
for i in (1,2,3,4,5):
        print(i)
```

Output

```
1
2
3
4
5
```

Lists and tuples have a special keyword (**len**) that can be use to find out the size of a list/tuple:

Python 2.x / 3.x

```
x = [1,2,3,4,5]
y = (10,20,300)
print (len(x), len(y))
```

Output 3.x

```
5 3
```

Lists and functional programming

A list can also be build using functional programming.

- A list of numbers from 1 to 9

Python 2.x / 3.x

```
x = [i for i in range(1,10)] #x = [1,2,3,4,5,6,7,8,9]
```

- A list of all divisor of 23 smaller than 100

Python 2.x / 3.x

```
x = [i for i in range(1,100) if i % 23 == 0] #x = [23, 46, 69, 92]
```

- A list of all square values for number from 1 to 5

Python 2.x / 3.x

```
x = [i*i for i in range(1,6)] #x = [1, 4, 9, 16, 25]
```

Lists and functional programming

A list can also be build using functional programming.

- A list of pairs of numbers from 1 to 10 that summed up produce a number that divides with 7

Python 2.x / 3.x

```
x=[[x, y] for x in range(1,10) for y in range(1,10) if (x+y)%7==0]
#x = [[1, 6], [2, 5], [3, 4], [4, 3], [5, 2], [5, 9], [6, 1],
#     [6, 8], [7, 7], [8, 6], [9, 5]]
```

- A list of tuples of numbers from 1 to 10 that summed up produce a number that divides with 7

Python 2.x / 3.x

```
x=[(x, y) for x in range(1,10) for y in range(1,10) if (x+y)%7==0]
#x = [(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (5, 9), (6, 1),
#     (6, 8), (7, 7), (8, 6), (9, 5)]
```

Lists and functional programming

A list can also be build using functional programming.

- A list of prime numbers that a smaller than 100

Python 2.x / 3.x

```
x=[x for x in range(2,100) if len([y for y in range(2,x//2+1) if x % y==0])==0]
#x = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
      59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Using functional programming in Python drastically reduces the size of code. However, depending on how large the expression is to build a list, functional programming may not be advisable if your aim is readability.

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Add a new element in the list (either use the member function(method) **append** or the operator **+=**). To add lists or tuples use **extend** method

Python 2.x / 3.x

```
x = [1,2,3]           #x = [1, 2, 3]
x.append(4)           #x = [1, 2, 3, 4]
x+=[5]                #x = [1, 2, 3, 4, 5]
x+=[6,7]              #x = [1, 2, 3, 4, 5, 6, 7]
x+=(8,9,10)           #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x[len(x):] = [11]     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
x.extend([12,13])     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
x.extend((14,15))     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                      #     14,15]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieved by using some operators.

- Insert a new element in the list using member function(method) **insert**

Python 2.x / 3.x

```
x = [1,2,3]              #x = [1, 2, 3]
x.insert(1,"A")          #x = [1, "A", 2, 3]
x.insert(-1,"B")         #x = [1, "A", 2, "B", 3]
x.insert(len(x),"C")     #x = [1, "A", 2, "B", 3, "C"]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Insert a new element or multiple elements can be done using [:] operator. Similarly [] operator can be used to change the value of one element

Python 2.x / 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
x[2] = 20                #x = [1, 2, 20, 4, 5]
x[3:] = ["A","B","C"]    #x = [1, 2, 20, "A", "B", "C"]
x[:4] = [10]             #x = [10, "B", "C"]
x[1:3] = ['x','y','z']   #x = [10, "x", "y", "z"]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Remove an element in the list ➔ using member function(method) **remove**. This method removes the first element with a given value

Python 2.x / 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
x.remove(1)                 #x = [2, 3]
x.remove(100)               #!!! ERROR !!! - 100 is not a value from x
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To remove an element from a specific position the **del** keyword can be used.

Python 2.x / 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[2]                 #x = [1, 2, 4, 5]
del x[-1]                #x = [1, 2, 4]
del x[0]                 #x = [2, 4]
del x[1000]              #!!! ERROR !!! - 1000 is not a valid index
```

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[4:]                #x = [1, 2, 3, 4]
del x[:2]                #x = [3, 4]
```

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[2:4]               #x = [1, 2, 5]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To **pop** method can be use to remove an element from a desire position an return it. This method can be use without any parameter (and in this case it refers to the last element)

Python 2.x / 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
y = x.pop(2)             #x = [1, 2, 4, 5]      y = 3
y = x.pop(0)             #x = [2, 4, 5]         y = 1
y = x.pop(-1)            #x = [2, 4]            y = 5
y = x.pop()              #x = [2]               y = 4
y = x.pop(1000)          #!!! ERROR !!! - 1000 is not a valid index
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To clear the entire list the **del** command can be used

Python 2.x / 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[:]                 #x = []
```

- Python 3.x also has a method **clear** that can be used to clear an entire list

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
x.clear()                #x = []
```

Lists

Be aware that using the operator (=) does not make a copy but only a reference of a list.

Python 2.x / 3.x

```
x = [1,2,3]
y = x
y.append(10)
#x = [1,2,3,10]
#y = [1,2,3,10]
```

If you want to make a copy of a list, use the **list** keyword:

Python 2.x / 3.x

```
x = [1,2,3]
y = list (x)
y.append(10)
#x = [1,2,3]
#y = [1,2,3,10]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Python 3.x also has a method **copy** that can be used to create a shallow copy of a list

Python 3.x

```
x = [1,2,3]              #x = [1, 2, 3]
b = x.copy()             #x = [1, 2, 3]    b = [1, 2, 3]
b += [4]                 #x = [1, 2, 3]    b = [1, 2, 3, 4]
```

- The operator [:] can also be use to achieve the same result

Python 2.x/3.x

```
x = [1,2,3]              #x = [1, 2, 3]
b = x[:]                 #x = [1, 2, 3]    b = [1, 2, 3]
b += [4]                 #x = [1, 2, 3]    b = [1, 2, 3, 4]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **index** method to find out the position of a specific element in a list

Python 2.x/3.x

```
x = ["A","B","C","D"] #x = [”A”, ”B”, ”C”, ”D”, ”E”]
y = x.index("C")      #y = 2
y = x.index("Y")      #!!! ERROR !!! – ”Y” is not part of list x
```

- The operator **in** can be used to check if an element exists in the list

Python 2.x/3.x

```
x = ["A","B","C","D"] #x = [”A”, ”B”, ”C”, ”D”, ”E”]
y = "C" in x          #y = True
y = "Y" in x          #y = False
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **count** method to find out how many elements of a specific value exists in a list

Python 2.x/3.x

```
x = [1,2,3,2,5,3,1,2,4,2] #x = [1,2,3,2,5,3,1,2,4,2]
y = x.count(2)            #y = 4 [1,2,3,2,5,3,1,2,4,2]
y = x.count(0)            #y = 0
```

- The **reverse** method can be used to reverse the elements order from a list

Python 2.x/3.x

```
x = [1,2,3]               #x = [1,2,3]
x.reverse ()              #x = [3,2,1]
```

Lists

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **sort** method to sort elements from the list

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

Python 2.x/3.x

```
x = [2,1,4,3,5]
x.sort()                                        #x = [1,2,3,4,5]
x.sort(reverse=True)                            #x = [5,4,3,2,1]
x.sort(key = lambda i: i%3)                     #x = [3,1,4,2,5]
x.sort(key = lambda i: i%3,reverse=True)        #x = [2,5,1,4,3]
```

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **map** to create a new list where each element is obtained based on the lambda expression provided.

map (*function, iterableElement$_1$, [iterableElement$_2$,... iterableElement$_n$]* **)**

Python 2.x/3.x

```
x = [1,2,3,4,5]
y = list(map(lambda element: element*element,x))   #y = [1,4,9,16,25]
```

```
x = [1,2,3]
y = [4,5,6]
z = list(map(lambda e1,e2: e1+e2,x,y))             #z = [5,7,9]
```

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- **map** function returns a list in Python 2.x and an iterable parameter in Python 3.x

Python

```
x = [1,2,3]
y = map(lambda element: element*element,x)
#y = [1,4,9]              →Python 2.x
#y = iterable object      →Python 3.x
```

- **map** function returns a list in Python 2.x and an iterable parameter in Python 3.x

Python

```
x = [1,2,3]
y = [4,5,6,7]
z = list(map(lambda e1,e2: e1+e2,x,y))   #z = [5,7,9] → Python 3.x
                                         #!!! ERROR on→ Python 2.x
```

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **filter** to create a new list where each element is filtered based on the lambda expression provided.

Filter (*function, iterableElement* **)**

Python 2.x/3.x

```
x = [1,2,3,4,5]
y = list(filter(lambda element: element%2==0,x))   #y = [2,4]
```

- Just like in the case of **map** function, **filter** function has different results in Python 2.x and Python 3.x: Python 2.x (returns a list), in Python 3.x returns an iterable object (a filtered object)

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Both **filter** and **map** can also be used to create a list (usually in conjunction with **range** keyword)

Python 2.x/3.x

```
x = list(map(lambda x: x*x, range(1,10)))
#x = [1, 4, 9, 16, 25, 36, 49, 64, 81]

x = list(filter(lambda x: x%7==1,range(1,100)))
#x = [1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99]
```

- Python 2.x had another function (**reduce**) that was removed from Python 3.x

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **min** and **max** functions to find out the biggest/smallest element from an iterable list based on the lambda expression provided.

max (*iterableElement, [key]*) **max** (el_1, el_2, … *[key]*)	**min** (*iterableElement, [key]*) **min** (el_1, el_2, … *[key]*)

Python 2.x/3.x

```
x = [1,2,3,4,5]
y = max (x)                                  #y = 5
y = max (1,3,2,7,9,3,5)                      #y = 9
y = max (x,key = lambda i: i % 3)            #y = 2
```

- If you want to use a **key** for max and/or min function, be sure that you added with the parameter name decoration: key = <function>, and not just the key_function or a lambda.

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **sum** to add all elements from an iterable object. Elements from the iterable objects should allow the possibility of addition with other elements.

sum (iterableElement, *[startValue]*)

- *startValue* represent the value from where to start summing the elements. Default is 0

Python 2.x/3.x

```
x = [1,2,3,4,5]
y = sum (x)                    #y = 15
y = sum (x,100)                #y = 115 (100+15)
```

```
x = [1,2,"3",4,5]
y = sum (x)                    #ERROR→ Can't add int and string
```

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **sorted** to sort the element from a list (iterable object). The key in this case represents a compare function between two elements of the iterable object.

sorted (iterableElement, *[key],[reverse]*)

- The *reverse* parameter if not specified is considered to be False

Python 2.x/3.x

```
x = [2,1,4,3,5]
y = sorted (x)                                  #y = [1,2,3,4,5]
y = sorted (x,reverse=True)                     #y = [5,4,3,2,1]
y = sorted (x,key = lambda i: i%3)              #y = [3,1,4,2,5]
y = sorted (x,key = lambda i: i%3,reverse=True) #y = [2,5,1,4,3]
```

- Just like in the precedent case, you have to use the optional parameter with their name

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **reversed** to reverse the element from a list (iterable object).

Python 2.x/3.x

```
x = [2,1,4,3,5]
y = list (reversed(x))                    #y = [5,3,4,1,2]
```

- Use **any** and **all** to check if at least one or all elements from a list (iterable objects) can be evaluated to true.

Python 2.x/3.x

```
x = [2,1,0,3,5]
y = any(x)     #y = True, all numbers except 0 are evaluated to True
y = all(x)     #y = False, 0 is evaluated to False
```

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **zip** to group 2 or more iterable objects into one iterable object

Python 2.x/3.x

```
x = [1,2,3]
y = [10,20,30]
z = list(zip(x,y))   #z = [(1,10) , (2,20) , (3,30)]
```

- Use **zip** with * character to unzip such a list. The unzip variables are tuples

Python 2.x/3.x

```
x = [(1,2) , (3,4) , (5,6)]
a,b = zip(*x)          #a = (1,3,5) and b = (2,4,6)
```

Built-in functions for list

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **del** to delete a list or a tuple

Python 2.x/3.x

```
x = [1,2,3]
del x
print (x)      #!!!ERROR!!! x no longer exists
```

Programming in Python

GAVRILUT DRAGOS

COURSE 3

Sets

A list of unique data (two elements **a** and **b** are considered unique if **a** is different than **b** ➔ this translates as **a** is of a different type than **b** or if **a** and **b** are of the same type, that ***a != b***)

A special keyword **set** can be used to create a set. The **{** and **}** can also be used to build a set. Set keyword can be used to initialize a set from tuples ,lists or strings.

Sets supports some special mathematical operations like:

- Intersection
- Union
- Difference
- Symmetric difference

Sets

Python 2.x / 3.x

```
x = set()                  #x is an empty set

x = {1,2,3}                #x is a set containing 3 elements: 1,2 and 3
x = {1,2,2,3,1,1}          #x is a set containing 3 elements: 1,2 and 3
x = {1,2,"AB","ab"}        #x is a set containing 4 elements: 1,2,"AB" and "ab"
x = set((1,2,3,2))         #x is a set containing 3 elements: 1,2 and 3
x = set([1,2,3,2])         #x is a set containing 3 elements: 1,2 and 3
x = set("Hello")           #x is a set containing 4 characters: H,e,l and o
```

Sets

Elements from a set can NOT be accessed (they are unordered collections):

Python 2.x / 3.x

```
x = {'A', 'B', 2, 3, 'C'}
x[0], x[1], x[1:2], … ➔ all this expression will produce an error
```

Similarly – there is no addition operation defined between two sets:

Python 2.x / 3.x

```
x = {'A', 'B', 2, 3, 'C'}
y = {'D', 'E', 1}
z = y + z                    #!!!ERROR !!
```

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Add a new element in the set (either use the member function(method) **add**)

Python 2.x / 3.x

```
x = {1,2,3}          #x = {1, 2, 3}
x.add(4)             #x = {1, 2, 3, 4}
x.add(1)             #x = {1, 2, 3, 4}
```

- Remove a element from the set (methods **remove** or **discard**). Remove throws an error if the set does not contain that element. Use **clear** method to empty an entire set.

Python 2.x / 3.x

```
x = {1,2,3}          #x = {1, 2, 3}
x.remove(1)          #x = {2, 3}
x.discard(2)         #x = {3}
x.discard(2)         #x = {3}
```

```
x = {1,2,3}          #x = {1, 2, 3}
x.clear()            #x = {}
```

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Several elements can be added to a set by either use the member function(method) **update** or by using the operator **|=**

Python 2.x / 3.x

```
x = {1,2,3}                 #x = {1, 2, 3}
x |= {3,4,5}                #x = {1, 2, 3, 4, 5}
x.update({5,6})             #x = {1, 2, 3, 4, 5, 6}
x.update({5,6},{6,7})       #x = {1, 2, 3, 4, 5, 6, 7}
x.update({8},{6},{9})       #x = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- **update** method can be called with multiple parameters (sets)

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Union operation can be performed by using the operator | or the method **union**

Python 2.x / 3.x

```
x = {1,2,3}
y = {3,4,5}
t = {2,4,6}
z = x | y | t              #z = {1, 2, 3, 4, 5, 6}
s = {7,8}
w = x.union(s)             #w = {1, 2, 7, 8}
w = x.union(s, y, t)       #w = {1, 2, 3, 4, 5, 6, 7, 8}
```

- **union** method can be called with multiple parameters (sets)

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Intersection operation can be performed by using the operator & or the method **intersection**

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {2,3,4,5}
t = {3,4,5,6}
z = x & y & t                #z = {3, 4}
w = x.intersection(y)        #w = {2, 3, 4}
w = x.intersection(y, t)     #w = {3, 4}
```

- **intersection** method can be called with multiple parameters (sets)

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Difference operation can be performed by using the operator - or the method **difference**

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {2,3,4,5}
z = x - y                  #z = {1}
z = y - x                  #z = {5}
w = x.difference(y)        #w = {1}
s = {1,2,3}
w = x.difference(y,s)      #w = {} → empty set
```

- **difference** method can be called with multiple parameters (sets)

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Symmetric difference operation can be performed by using the operator ^ or the method **symmetric_difference**

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {2,3,4,5}
z = x ^ y                          #z = {1, 5}
z = y ^ x                          #z = {1, 5}
w = x.symmetric_difference(y)      #w = {1, 5}
s = {1,2,3}
w = x.symmetric_difference(y,s)    #!!! ERROR !!!
```

- **symmetric_difference** method can **NOT** be called with multiple parameters (sets)

Sets

All sets operations also support some operations that apply to one variable such as:

- Intersection
  - **intersection_update**
  - **&=**
- Difference
  - **difference_update**
  - **-=**
- Symmetric difference
  - **symmetric_difference_update**
  - **^=**

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- To test if a an element exists in a set, we can use the **in** operator

Python 2.x / 3.x

```
x = {1,2,3,4}
y = 2 in x                    #y = True
z = 5 not in x                #z = True
```

- The length of a set can be found out using the **len** keyword

Python 2.x / 3.x

```
x = {10,20,30,40}
y = len (x)                   #y = 4
```

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Use method **isdisjoint** to test if a set has no common elements with another one

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {10,20,30,40}
z = x.isdisjoint(y)                    #z = True
```

- Use method **issubset** or operator **<=** to test if a set is included in another one

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {1,2,3,4,5,6}
z = x.issubset(y)                      #z = True
t = x <= y                             #t = True
```

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Use method **issuperset** or operator **>=** to test if a set is included in another one

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {1,2,3,4,5,6}
z = y.issuperset(x)                    #z = True
t = y >= x                             #t = True
```

- Operator > can also be used → it checks if a set is included in another **BUT** is not identical to it. Operator < can be used in the same way.

Python 2.x / 3.x

```
x = {1,2,3,4}
y = {1,2,3,4,5,6}
t = y > x                              #t = True
```

```
x = {1,2,3,4}
y = {1,2,3,4}
t = y > x                              #t = False
```

Sets

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Use method **pop** to remove one element from the set. The remove element is different from Python 2.x to Python 3.x in terms of the order the element are kept in memory. Even if sets are unordered collection, in order to have quick access to different elements of the set these elements must be kept in memory in a certain way.

Python 2.x / 3.x

```
x = {"A","a","B","b",1,2,3}
print (x)
print (x.pop())
```

Output (Python 3)

```
{1, 2, 3, 'b', 'B', 'A', 'a'}
1
```

Output (Python 2)

```
set(['a', 1, 2, 3, 'b', 'A', 'B'])
a
```

- Use **copy** method to make a shallow copy of a set.

Sets and functional programming

A set can also be built using functional programming

- The main difference is that instead of operator [...] to build a set one need to use {...}

Python 2.x / 3.x

```
x = {i for i in range(1,9)}                          #x = {1,2,3,4,5,6,7,8}
x = {i for i in range(1,100) if i % 23 == 0}         #x = {23, 46, 69, 92}
x = {i*i for i in range(1,6)}                        #x = {1, 4, 9, 16, 25}
```

- The condition of the set (all elements are unique) still applies

Python 2.x / 3.x

```
x = {i%5 for i in range(1,100)}                      #x = {0, 1, 2, 3, 4|
```

Sets and Built-in functions

The default build-in functions for list can also be used with sets and lambdas.

- Use **map** to create a new set where each element is obtained based on the lambda expression provided.

Python 2.x/3.x

```
x = {1,2,3,4,5}
y = set(map(lambda element: element*element,x))    #y = {1,4,9,16,25}
```

```
x = [1,2,3]
y = [4,5,6]
z = set(map(lambda e1,e2: e1+e2,x,y))              #z = {5,7,9}
```

Sets and Built-in functions

The default build-in functions for list can also be used with sets and lambdas.

- Use **filter** to create a new set where each element is filtered based on the lambda expression provided.

Python 2.x/3.x

```
x = [1,2,3,4,5]
y = set(filter(lambda element: element%2==0,x))      #y = {2,4}
```

- Both **filter** and **map** are used to create a set (usually in conjunction with **range** keyword)

Python 2.x/3.x

```
x = set(map(lambda x: x*x, range(1,10)))
#x = {1, 4, 9, 16, 25, 36, 49, 64, 81}

x = set(filter(lambda x: x%7==1,range(1,100)))
#x = {1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99}
```

Sets and Built-in functions

The default build-in functions for list can also be used with sets and lambdas.

- Other functions that work in a similar way as the build-in functions for list are: **min**, **max**, **sum**, **any**, **all, sorted, reversed**
- **for** statement can also be used to enumerate between elements of a set

Python 2.x / 3.x

```
for i in {1,2,3,4,5}:
        print(i)
```

- Python language also has another type → *frozenset*. A frozen set has all of the characteristics of a normal set, but it can not be modified. To create a frozen set use the **frozenset** keyword.

Python 2.x/3.x

```
x = frozenset ({1,2,3})
x.add(10)                                    #!!!ERROR!!!
```

Dictionaries

A dictionary is python implementation of a hash-map container. Design as a (key – value pair) where Key is a unique element within the dictionary.

A special keyword **dict** can be used to create a dictionary. The **{** and **}** can also be used to build a dictionary – much like in the case of sets.

Python 2.x / 3.x

```
x = dict()                                  #x is an empty dictionary
x = {}                                      #x is an empty dictionary
x = {"A":1, "B":2}                          #x is a dictionary with 2 keys
                                            #("A" and "B")
x = dict(abc=1,aaa=2)                       #equivalent to x= {"abc":1,  "aaa":2}
x = dict({"abc":1,"aaa":2})                 #equivalent to x= {"abc":1,  "aaa":2}
x = dict([("abc",1) ,("aaa",2)])            #equivalent to x= {"abc":1,  "aaa":2}
x = dict((("abc",1) ,("aaa",2)))            #equivalent to x= {"abc":1,  "aaa":2}
x = dict(zip(["abc","aaa"],[1,2]))#equivalent to x= {"abc":1,  "aaa":2}
```

Dictionaries

To set a value in a dictionary use [] operator. The same operator can be used to read an existing value. If a value does not exists, an exception will be thrown.

Python 2.x / 3.x

```
x = {}                          #x is an empty dictionary
x["ABC"] = 2                    #x is an dictionary with one key (ABC)
y = x["ABC"]                    #y = 2
y = x["test"]                   #!!! ERROR !!!
```

To check if a key exists in a dictionary, use **in** operator. **len** can also be used to find out how may keys a dictionary has.

Python 2.x / 3.x

```
x = {"A":1, "B":2}              #x is a dictionary with 2 keys
"A" in x                        #True
len (x)                         #2
```

Dictionaries

Values from a dictionary can also be manipulated with **setdefault** member.

Python 2.x / 3.x

```
x = {"A":1, "B":2}          #x = {"A":1,"B":2}
y = x.setdefault("C",3)     #x = {"A":1,"B":2,"C:3"},              y=3
y = x.setdefault("D")       #x = {"A":1,"B":2,"C:3","D":None},     y=None
y = x.setdefault("A")       #x = {"A":1,"B":2,"C:3","D":None},     y=1
y = x.setdefault("B",20)    #x = {"A":1,"B":2,"C:3","D":None},     y=2
```

Method **update** can also be used to change the value associated with a key.

Python 2.x / 3.x

```
x = {"A":1, "B":2}          #x = {"A":1,"B":2}
x.update({"A":10})          #x = {"A":10,"B":2}
x.update({"A":100,"B":5})   #x = {"A":100,"B":5}
x.update({"C":3})           #x = {"A":100,"B":5,"C":3}
x.update(D=123,E=111)       #x = {"A":100,"B":5,"C":3,"D":123,"E":111}
```

Dictionaries

To delete an element from a dictionary use **del** keyword or **clear** method

Python 2.x / 3.x

```
x = {"A":1, "B":2}                  #x = {"A":1,"B":2}
del x["A"]                          #x = {"B":2}
x.clear()                           #x is an empty dictionary
del x["C"]                          #!!! ERROR !!! "C" is not a key in x
```

To create a new dictionary you can use **copy** or static method **fromkeys**

Python 2.x / 3.x

```
x = {"A":1, "B":2}                  #x={"A":1,"B":2}
y = x.copy()                        #makes a shallow copy of x
y["C"]=3                            #x={"A":1,"B":2},y={"A":1,"B":2,"C":3}
```

```
x = dict.fromkeys(["A","B"])        #x = {"A":None,"B":None}
x = dict.fromkeys(["A","B"],2)      #x = {"A":2,"B":2}
```

Dictionaries

Elements from the dictionary can also be accessed with method **get**

Python 2.x / 3.x

```
x = {"A":1, "B":2}              #x = {"A":1,"B":2}
y = x.get("A")                  #y = 1
y = x.get("C")                  #y = None
y = x.get("C",123)              #y = 123
```

An element can also be extracted using pop method.

Python 2.x / 3.x

```
x = {"A":1, "B":2}              #x={"A":1,"B":2}
y = x.pop("A")                  #x={"B":2}, y = 1
y = x.pop("C",123)              #x={"B":2}, y = 123
y = x.pop("D")                  #!!! ERROR !!! Key "D" does not exists
                                #and no default value was provided
```

Dictionaries and functional programming

A dictionary can also be built using functional programming

Python 2.x / 3.x

```
x = {i:i for i in range(1,9)}
#x = {1:1,2:2,3:3,4:4,5:5,6:6,7:7,8:8}
```

```
x = {i:str(64+i) for i in range(1,9)}
#x = {1:"A",2:"B",3:"C",4:"D",5:"E",6:"F",7:"G",8:"H"}
```

```
x = {i%3:i for i in range(1,9)}
#x = {0:6,1:7,2:8} → last values that were updated
```

```
x = {i:str(64+i) for i in range(1,9) if i%2==0}
#x = {2:"B", 4:"D", 6:"F", 8:"H"}
```

Dictionaries

Keys from the dictionary can be obtained with method keys

Python 2.x / 3.x

```
x = {"A":1, "B":2}                    #x = {"A":1,"B":2}
y = x.keys()                          #y = ["A","B"] → an iterable object
```

To iterate all keys from a dictionary:

Python 2.x / 3.x

```
x = {"A":1, "B":2}
for i in x:
        print (i)
```

```
x = {"A":1, "B":2}
for i in x.keys():
        print (i)
```

Output

```
A
B
```

Dictionaries

Values from the dictionary can be obtained with method **values**

Python 2.x / 3.x

```
x = {"A":1, "B":2}                    #x = {"A":1,"B":2}
y = x.values()                        #y = ["1","2"] → an iterable object
```

To iterate all values from a dictionary:

Python 2.x / 3.x

```
x = {"A":1, "B":2}
for i in x.values():
        print (i)
```

Output

```
1
2
```

Output order may be different for different versions of python depending on how data is stored/ordered in memory.

Dictionaries

All pairs from a dictionary can be obtained using the method **items**

Python 2.x / 3.x

```
x = {"A":1, "B":2}                  #x = {"A":1,"B":2}
y = x.items()                       #y = an iterable object (Python 3) or
                                    #a list of tuples for Python 2.
                                    #[ ("A":1) , ("B":2) ]
```

To iterate all keys from a dictionary:

Python 2.x / 3.x

```
x = {"A":1, "B":2}
for i in x.items():
        print (i)
```

Output

("A", 1)
("B", 2)

Dictionaries

Using the **items** method elements from a dictionary can be sorted according to their value.

Python 2.x / 3.x

```
x = {
        "Dacia"  : 120,
        "BMW"    : 160,
        "Toyota" : 140
    }
for i in sorted(x.items(),key = lambda element : element[1]):
        print (i)
```

Output

("Dacia", 120)
("Toyota", 140)
("BMW", 160)

Dictionaries

Operator ** can be used in a function to specify that the list of parameters of that function should be treated as a dictionary.

Python 2.x / 3.x

```
def GetFastestCar(**cars):
        min_speed = 0
        name = None
        for car_name in cars:
                if cars[car_name] > min_speed:
                        name = car_name
                        min_speed = cars[car_name]
        return name
fastest_car = GetFastestCar(Dacia=120,BMW=160,Toyota=140)
print (fastest_car)
#fastes_car = ”BMW”
```

Dictionaries

Build-in functions such as **filter** can also be used with dictionaries.

Python 2.x / 3.x

```
x = {
        "Dacia"  : 120,
        "BMW"    : 160,
        "Toyota" : 140
    }
y = dict(filter(lambda element : element[1]>=140,x.items()))
#y = {"Toyota":140, "BMW":160}
```

To delete an entire dictionary use **del** keyword.

Programming in Python

GAVRILUT DRAGOS

COURSE 4

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except:
    #code that will be executed in
    #case of any exception
```

Python 2.x / 3.x

```
try:
        x = 5 / 0
except:
        print("Exception")
```

Output

```
Exception
```

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except:
    #code that will be executed in
    #case of any exception
else:
    #code that will be executed if
    #there is no exception
```

Python 2.x / 3.x

```
try:
        x = 5 / 1
except:
        print("Exception")
else:
        print(“All ok")
```

Output

```
All ok
```

Exceptions

All exceptions in python are derived from **BaseException** class. There are multiple types of exceptions including: **ArithmeticError**, **BufferError**, **AttributeError**, **FloatingPointError**, **IndexError**, **KeyboardInterrupt**, **NotImplementedError**, **OverflowError**, **IndentationError**, and many more.

A list of all the exceptions can be found on:

- https://docs.python.org/3.5/library/exceptions.html#Exception
- https://docs.python.org/2.7/library/exceptions.html#Exception

A custom (user-defined) exception type can also created (more on this topic at “Classes”).

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except ExceptionType1:
    #code for exception of type 1
except ExceptionType2:
    #code for exception of type 1
except:
    #code for general exception
else:
    #code that will be executed if
    #there is no exception
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except ArithmeticError:
        print("ArithmeticError")
    except:
        print("Generic exception")
    else:
        print("All ok")

Test(0)
Test("aaa")
Test(1)
```

Output

```
ArithmeticError
Generic exception
All ok
```

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except ExceptionType1:
    #code for exception of type 1
except ExceptionType2:
    #code for exception of type 1
except:
    #code for general exception
else:
    #code that will be executed if
    #there is no exception
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Generic exception")
    except ArithmeticError:
        print("ArithmeticError")
    else:
        print("All ok")

Test(0)
Test("aaa")
Test(1)
```

Generic exception must be the last one. Code will not compile.

Exceptions

Python also have a finally keyword that can be use to executed something at the end of the try block.

Python 2.x / 3.x

```
try:
    #code
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
finally:
    #code that will be executed
    #after the try block execution
    #is completed
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Error")
    else:
        print("All ok")
    finally:
        print("Final")
Test(0)
Test(1)
```

Output

Test(0):
Error
Final

Test(1):
All ok
Final

Exceptions

Python also have a finally keyword that can be use to executed something at the end of the try block.

Python 2.x / 3.x

```
try:
    #code
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
finally:
    #code that will be executed
    #after the try block execution
    #is completed
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Error")
    finally:
        print("Final")
    else:
        print("All ok")
Test(0)
Test(1)
```

Finally must be the last statement

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except (Type1,Type2,…Typen):
    #code for exception of type
    #1,2,…
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except (ArithmeticError,TypeError):
        print("ArithmeticError")
    except:
        print("Generic exception")
    else:
        print("All ok")

Test(0)
Test("aaa")
Test(1)
```

Output

```
ArithmeticError
ArithmeticError
All ok
```

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except Type1 as <var_name>:
    #code for exception of type
    #1.
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
```

Python 2.x / 3.x

```
try:
    x = 5 / 0
except Exception as e:
    print( str(e) )
```

Output

Python2: integer division or modulo by zero
Python3: division by 0

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except (Type1,Type2,…Typen) as <var>:
    #code for exception of type 1,2,… n
```

```
try:
    x = 5 / 0
except (Exception,ArithmeticError,TypeError) as e:
    print( str(e), type(e) )
```

Output

Python2: ('integer division or modulo by zero', <type 'exceptions.ZeroDivisionError'>)
Python3: division by zero <class 'ZeroDivisionError'>

Exceptions

Python also has another keyword (**raise**) that can be use to create or throw an exception:

Python 2.x / 3.x

```
raise ExceptionType (parameters)
raise ExceptionType (parameters) from <exception_var>
```

```
try:
    raise Exception("Testing raise command")
except Exception as e:
    print(e)
```

Output

```
Testing raise command
```

Exceptions

Each exception has a list of arguments (parameter *args*)

Python 2.x / 3.x

```
try:
    raise Exception("Param1",10,"Param3")
except Exception as e:
    params = e.args
    print (len(params))
    print (params[0])
```

Output

```
3
Param1
```

Exceptions

raise keyword can be used without parameters. In this case it indicates that the current exception should be re-raised.

Python 2.x / 3.x

```
try:
    try:
        x = 5 / 0
    except Exception as e:
        print(e)
        raise
except Exception as e:
    print("Return from raise -> ",e)
```

Output (Python 3.x)

```
division by zero
Return from raise -> division by zero
```

Exceptions

Python 3.x supports chaining exception via **from** keyword.

Python 3.x

```
1  try:
2      x = 5 / 0
3  except Exception as e:
4      raise Exception("Error") from e
```

Output (Python 3.x)

```
Traceback (most recent call last):
  File "a.py", line 2, in <module>
    x = 5 / 0
ZeroDivisionError: division by zero
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "a.py", line 4, in <module>
    raise Exception("Error") from e
Exception: Error
```

Exceptions

Python has a special keyword (**assert**) that can be used to raise an exception based on the evaluation of a condition:

Python 2.x / 3.x

```
age = -1
try:
        assert (age>0),"Age should be a positive number"
except Exception as e:
        print (e)
```

Output

```
Age should be a positive number
```

Exceptions

pass keyword is usually used if you want to catch an exception but do not want to process it.

Python 2.x / 3.x

```
try:
        x = 10 / 0
except:
        pass
```

Some exceptions (if not handled) can be used for various purposes.

Python 2.x / 3.x

```
print("Test")
raise SystemExit
print("Test2")
```

This exception (SystemExit) if not handle will imediatelly terminate your program

Output

```
Test
```

Modules

Modules are python's libraries and extends python functionality. Python has a special keyword (**import**) that can be used to import modules.

Format (Python 2.x/3.x)

```
import module1,[module2,module3, … modulen]
```

Classes or items from a module can be imported separately using the following syntax.

Format (Python 2.x/3.x)

```
from module import object1,[object2,object3, … objectn]
from module import *
```

When importing a module aliases can also be made using "**as**" keyword

Format (Python 2.x/3.x)

```
import module1 as alias1,[module2 as alias2, … modulen as aliasn]
```

Modules

Python has a lot of default modules (**os**, **sys**, **re**, **math**, etc).

There is also a keyword (**dir**) that can be used to obtain a list of all the functions and objects that a module exports.

Format (Python 2.x/3.x)

```
import math
print dir(math)
```

Output (Python 3.x)

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
```

The list of functions/items from a module may vary from Python 2.x to Python 3.x and from version to version.

Modules

Python distribution modules:

- Python 3.x ➔ https://docs.python.org/3/py-modindex.html
- Python 2.x ➔ https://docs.python.org/2/py-modindex.html

Module	Purpose
collections	Implementation of different containers
ctype	Packing and unpacking bytes into c-like structures
datetime	Date and Time operators
email	Support for working with emails
hashlib	Implementation of different hashes (MD5, SHA, …)
json	JSON encoder and decoder
math	Mathematical functions
os	Different functions OS specific (make dir, delete files, rename files, paths, …)

Module	Purpose
re	Regular expression implementation
random	Random numbers
socket	Low-level network interface
subprocess	Processes
sys	System specific functions (stdin,stdout, arguments, loaded modules, …)
traceback	Exception traceback
urllib	Handling URLs / URL requests, etc
xml	XML file parser

Modules - sys

Python documentation page:

- Python 3.x ➔ https://docs.python.org/3/library/sys.html#sys.modules
- Python 2.x ➔ https://docs.python.org/2/library/sys.html#sys.modules

object	Purpose
sys.argv	A list of all parameters send to the python script
sys.platform	Current platform (Windows / Linux / MAC OSX)
sys.stdin sys.stdout, sys.stderr	Handlers for default I/O operations
sys.path	A list of strings that represent paths from where module will be loaded
sys.modules	A dictionary of modules that have been loaded

Modules - sys

sys.argv provides a list of all parameters that have been send from the command line to a python script. The first parameter is the name/path of the script.

File 'test.py' (Python 2.x/3.x)

```
import sys
print ("First parameter is",sys.argv[0])
```

Output

```
>>> python.exe C:\test.py
First parameter is C:\test.py
```

Modules - sys

Python 2.x/3.x (File: sum.py)

```
import sys
suma = 0
try:
    for val in sys.argv[1:]:
        suma += int(val)
    print("Sum=",suma)
except:
    print("Invalid parameters")
```

Output

```
>>> python.exe C:\sum.py 1 2 3 4
Sum = 10

>>> python.exe C:\sum.py 1 2 3 test
Invalid parameters
```

Modules - os

Python documentation page:

- Python 3.x ➔ https://docs.python.org/3/library/os.html
- Python 2.x ➔ https://docs.python.org/2/library/os.html

Includes functions for:

- Environment
- Processes (PID, Groups, etc)
- File system (change dir, enumerate files, delete files or directories, etc)
- File descriptor functions
- Terminal informations
- Process management (spawn processes, fork, etc)
- Working with file paths

Modules - os

Listing the contents of a folder (os.listdir → returns a list of child files and folders).

Python 2.x/3.x

```
import os
print (os.listdir(".."))
```

Output

['$Recycle.Bin', 'Android', 'Documents and Settings', 'Drivers', 'hiberfil.sys', 'Program Files', 'Program Files (x86)', 'ProgramData', 'Python27', 'Python35', 'System Volume Information', 'Users', 'Windows', …]

File and folder operations:

- os.mkdir / os.mkdirs → to create folders
- os.chdir → to change current path
- os.rmdir / os.removedirs → to delete a folder
- os.remove / os.unlink → to delete a file
- os.rename / os.renames → rename/move operations

Modules - os

os has a submodule (**path**) that can be use to perform different operations with file/directories paths.

Python 2.x/3.x

```
import os
print (os.path.join ("C:","Windows","System32"))
print (os.path.dirname ("C:\\Windows\\abc.txt"))
print (os.path.basename ("C:\\Windows\\abc.txt"))
print (os.path.splitext ("C:\\Windows\\abc.txt"))
print (os.path.exists ("C:\\Windows\\abc.txt"))
print (os.path.exists ("C:\\Windows\\abc.txt"))
print (os.path.isdir ("C:\\Windows"))
print (os.path.isfile ("C:\\Windows"))
print (os.path.isfile ("C:\\Windows\\abc.txt"))
```

Output

```
C:\Windows\System32
C:\Windows
abc.txt
["C:\Windows\abc", ".txt"]
False
True
False
False
```

Modules - os

Listing the contents of a folder recursively.

Python 2.x/3.x

```
import os

for (root,directories,files) in os.walk("."):
        for fileName in files:
                full_fileName = os.path.join(root,fileName)
                print (full_fileName)
```

Output

```
.\a
.\a.py
.\all.csv
.\run.bat
.\Folder1\version.1.6.0.0.txt
.\Folder1\version.1.6.0.1.txt
.\Folder1\Folder2\version.1.5.0.8.txt
```

os module can also be used to execute a system command or run an application via **system** function

Python 2.x/3.x

```
import os
os.system("dir *.* /a")
```

Input/Output

Python has 3 implicit ways to work with I/O:

A) IN: via keyboard (with **input** or **raw_input** keywords)

- There are several differences between python 2.x and python 3.x regarding reading from stdin

B) OUT: via **print** keyword

C) IN/OUT: via **open** keyword (to access files)

Input/Output

input keyword performs differently in Python 2.x and Python 3.x:

- In Python 2.x, the content read from the input is evaluated and returned
- In Python 3.x, the content read from the input is considered to be a string and returned

Format (Python 2.x/3.x)

```
input ()
input (message)
```

Python 2.x / 3.x

```
x = input("Enter: ")
print (x,type(x))
```

- Use **raw_input** in Python 2.x to obtain the same effect as in Python 3.x

Python 2.x	Python 3.x
>>> Enter: 10 (10, <type 'int'>)	>>> Enter: 10 10 <class 'str'>
>>> Enter: 1+2*3.0 (7.0, <type 'float'>)	>>> Enter: 1+2*3.0 1+2*3.0 <class 'str'>
>>> Enter: ”123” ('123', <type 'str'>)	>>> Enter: ”123” "123" <class 'str'>
>>> Enter: test !!!ERROR!!! (test can not be evaluated)	>>> Enter: test test <class 'str'>

Input/Output

print can be used to print a string in both Python 2 and Python 3. In Python 3 print is a function and supports multiple parameters:

Format (Python 3.x)
`print` (**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

Python 3.x	
>>> print (“test”) test	>>> print (“test”,10) test 10
>>> print (“test”,10,sep=“---”) test---10	>>> print (“test”);print(“test2”) test test2
>>> print (“test”,end=“***”);print(“test2”) test***test2	

File management

A file can be open in python using the keywork **open.**

Format (Python 3.x)

```
FileObject = open (filePath, mode='r', buffering=-1, encoding=None,
                   errors=None, newline=None, closefd=True, opener=None)
```

Format (Python 2.x)

```
FileObject = open (filePath, mode='r', buffering=-1)
```

Where mode is a combination of the following:

- "r" – read (default)
- "w" – write
- "x" – exclusive creation (fail if file exists)
- "a" – append
- "b" – binary mode
- "t" – text mode
- "+" – update (read and write)

File management

Python 3 also supports some extra parameters such as:

- encoding →if the file is open in text mode and you need translation from different encodings (UTF, etc)
- error → specify the way conversion errors for different encodings should be processed
- newline → also for text mode, specifies what should be consider a new line. If this value is set to None the character that is specific for the current operating system will be used

Documentation for open function:

- Python 3.x ➔ https://docs.python.org/3/library/functions.html#open
- Python 2.x ➔ https://docs.python.org/2/library/functions.html#open

File management

A file object has the following methods:

- f.close → closes current file
- f.tell → returns the current file position
- f.seek → sets the current file position
- f.read → reads a number of bytes from the file
- f.write → write a number of bytes into the file
- f.readline → reads a line from the file

Also – the file object is iterable and returns all text lines from a file.

Python 2.x/3.x

```
for line in open("a.py"):
        print (line.strip())
```

Lines read using this method contain the line-feed terminator. To remove it use strip or rstrip.

File management

Functional programming can also be used:

Python 2.x/3.x

```
x = [line for line in open("file.txt") if "Gen" in line.strip()]
print (len(x))
```

To read the entire content of the file in a buffer:

Python 2.x/3.x

```
data = open("file.txt","rb").read()
print (len(data))
print (data[0])
```

read method returns a string in Python 2.x and a buffer in Python 3.x → The output of the previous code will be a character (in Python 2.x) and a number representing the ascii code of that character in Python 3.x

To obtain a string in Python 3.x use “rt” instead of “rb”

File management

To create a file and write content in it:

Python 2.x/3.x

```
open("file.txt","wt").write("A new file ...")
```

It is a good policy to embed file operation in a try block

Python 2.x/3.x

```
try:
    f = open("abc.txt")
    for line in f:
        print(line.strip())
    f.close()
except:
    print("Unable to open file abc.txt")
```

File management

Once a file is open, the file object handle can be use to retrieve different information regarding that file:

Python 2.x/3.x

```
f = open("a.py","rb")
print ("File name        : ", f.name)
print ("File open mode : ", f.mode)
print ("Is it closed ? : ", f.closed)
```

Programming in Python

GAVRILUT DRAGOS

COURSE 5

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
```

Python 2.x / 3.x
File: test.py

```
import MyModule

print (MyModule.Sum(10,20))
```

Output

```
30
```

Both files test.py and MyModule.py are located in the same folder.

After the execution of test.py the following things will happen:

- *MyModule.pyc* file will appear in the same folder (Python 2.x)
- A folder with the name *__pycache__* that contains a file called *MyModule.cpython-35.pyc* will appear in the same folder (Python 3.5) → the version will be different for different versions of Python 3

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x / 3.x
File: test.py

```
import MyModule

print (MyModule.Sum(10,20))
import MyModule
```

Output

```
MyModule loaded
30
```

Loading a module will automatically execute any code (main code) that resides in that module.

The main code of a module (code that is written directly and not within a function or a class) will only me executed once (the first time a module is loaded).

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x / 3.x
File: test.py

```
import MyModule

print (MyModule.Sum(10,20))
import MyModule
```

What if MyModule iis not located in the same folder as test.py file ?

Output

```
Traceback (most recent call last):
  File “test.py", line 1, in <module>
    import sys,MyModule
ImportError: No module named 'MyModule'
```

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x / 3.x
File: test.py

```
import sys

sys.path += ["<folder>"]

import MyModule

print (MyModule.Sum(10,20))
import MyModule
```

Output

```
MyModule loaded
30
```

In the above piece of code "<folder>" represents a path to the folder where the file MyModule.py resides.

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x / 3.x
File: test.py

```
import MyModule

print (dir (MyModule))
```

Output

Python 2.x ➔ ['Sum', '__builtins__', '__doc__', '__file__', '__name__', '__package__']

Python 3.x ➔ ['Sum', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x / 3.x
File: test.py

```
import MyModule

print (MyModule.__file__)
print (MyModule.__name__)
print (MyModule.__package__)
```

Attributes:

- __file__ ➔ full path of the file that corresponds to the module (it could be a pyc file as well)
- __name__ ➔ name of the module (in this example : MyModule)
- __package__ ➔ name of the package (in this example empty string in Python 3.x and None in Python 2.x)

Modules

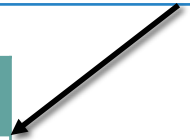
Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
print (__name__)
```

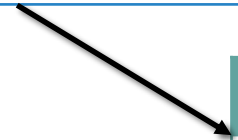
Python 2.x / 3.x
File: test.py

```
import MyModule
```



Output

__main__



Output

MyModule

If a python script is executed directly, the value of *__name__* parameter will be __main__. If it is executed using import, the value of *__name__* parameter will be the name of the module.

Modules

Any Python code (python script) can be used as a module.

Python 2.x / 3.x
File: MyModule.py

```
def Sum(x,y):
        return x+y
if __name__ == "__main__":
        print("Main code")
        print("Testing sum(10,20) = ",Sum(10,20))
else:
        print("Module loaded")
```

Python 2.x / 3.x
File: test.py

```
import MyModule
```

Output

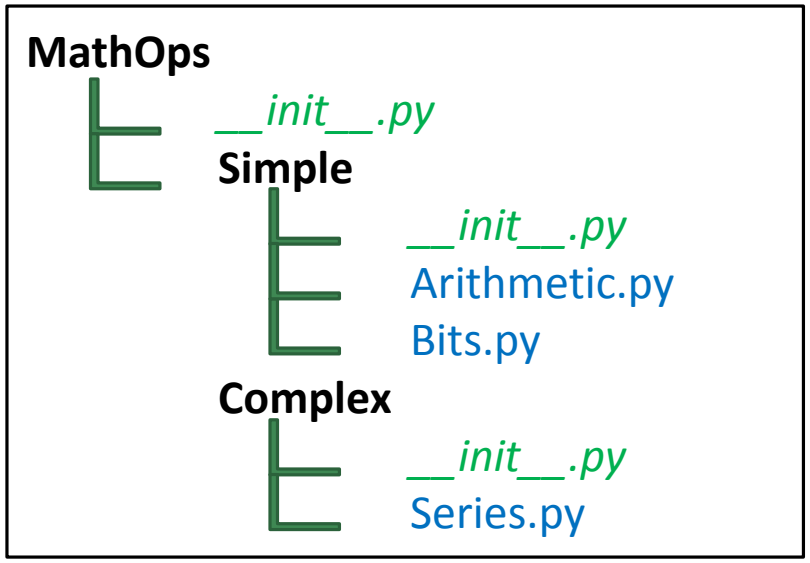
```
Main code
Testing sum(10,20) = 30
```

Output

```
Module loaded
```

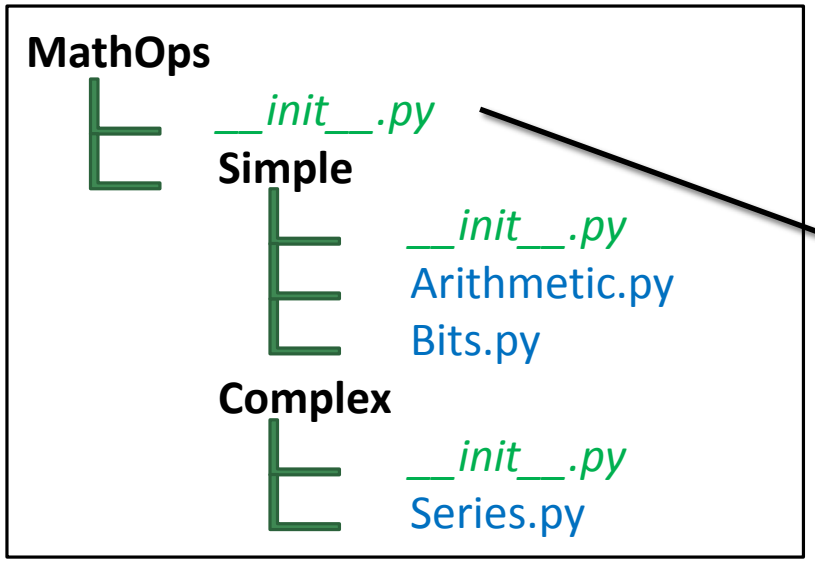
Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.



Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.

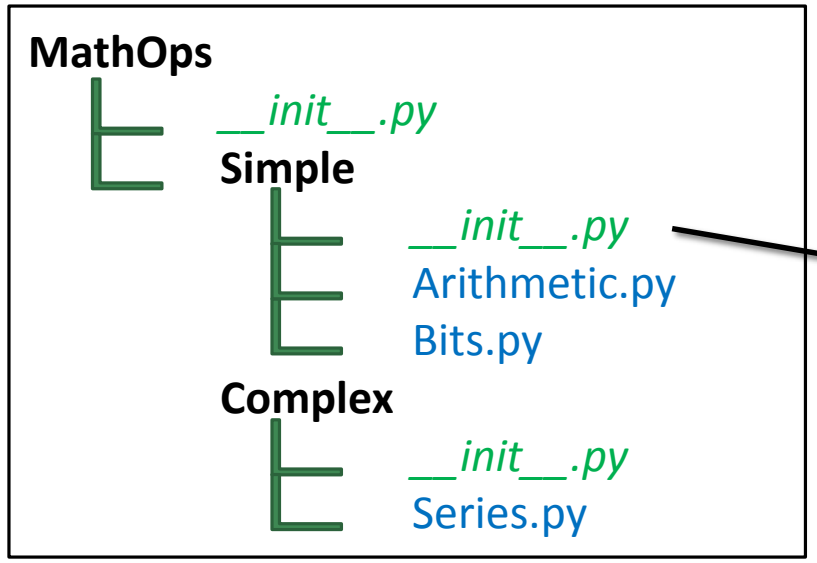


Python 2.x / 3.x
File: __init__.py

```
print ("Package MathOps init")
```

Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.

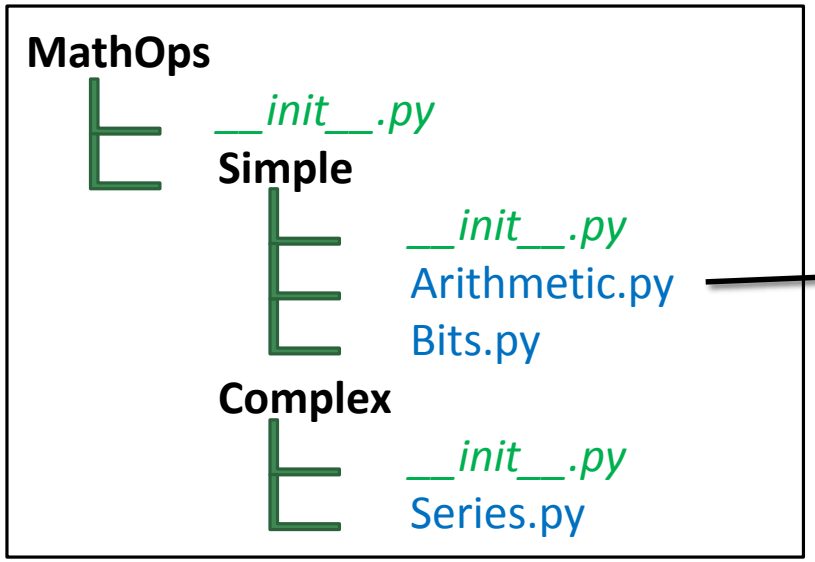


Python 2.x / 3.x
File: __init__.py

```
print ("Package MathOps.Simple init")
```

Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.

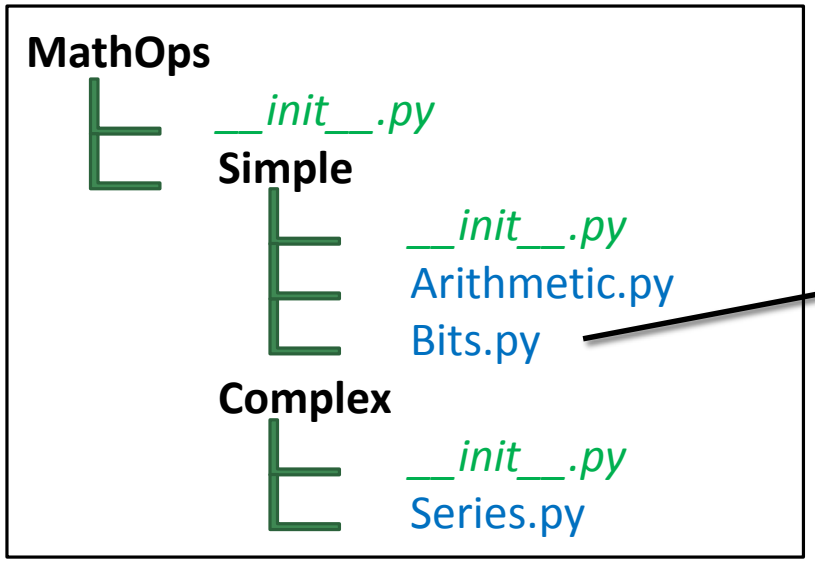


Python 2.x / 3.x
File: Arithmetic.py

```
def Add(x,y):
        return x+y
def Sub(x,y):
        return x-y
```

Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.

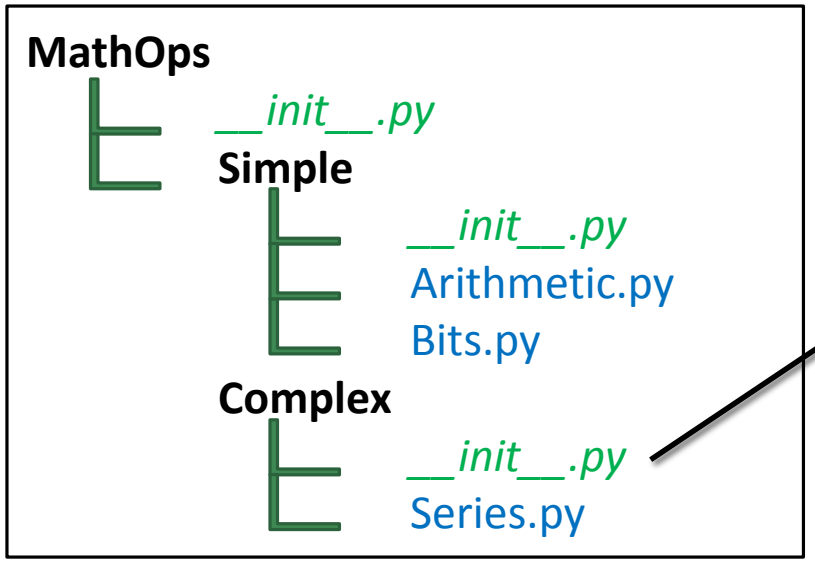


Python 2.x / 3.x
File: Bits.py

```
def SHL(x,y):
    return x << y
def SHR(x,y):
    return x >> y
```

Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.

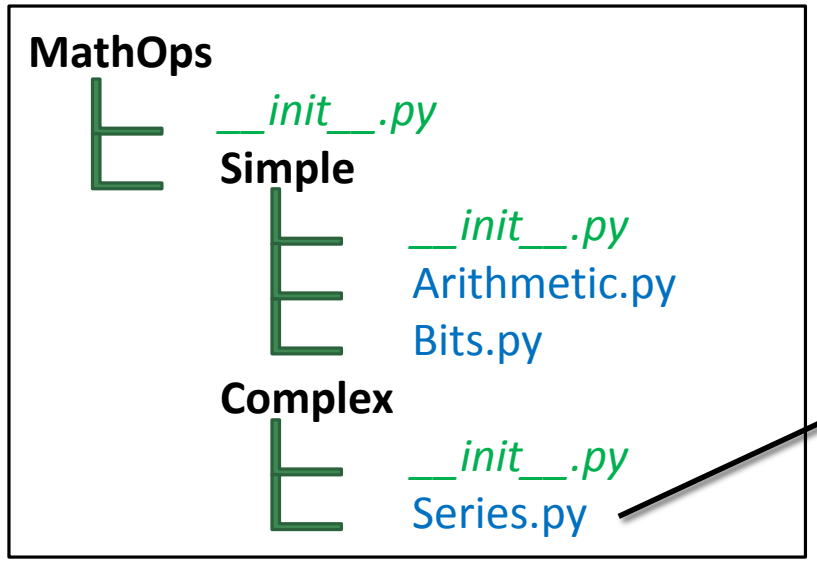


Python 2.x / 3.x
File: __init__.py

```
print ("Package MathOps.Complex init")
```

Packages

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a __init__.py must exist. That file is considered to be an entry point for that package/subpackage.



Python 2.x / 3.x
File: Series.py

```
def Sum(*p):
        c = 0
        for i in p:
                c+= i
        return c
def Product(*p):
        c = 1
        for i in p:
                c *= i
        return c
```

Packages

Usage:

Python 2.x / 3.x

```
import MathOps.Simple.Arithmetic

print (MathOps.Simple.Arithmetic.Add(2,3))
```

```
from MathOps.Simple import Arithmetic as a

print (a.Add(2,3))
```

Output

```
Package MathOps init
Package MathOps.Simple init
5
```

Packages

Usage:

Python 2.x / 3.x

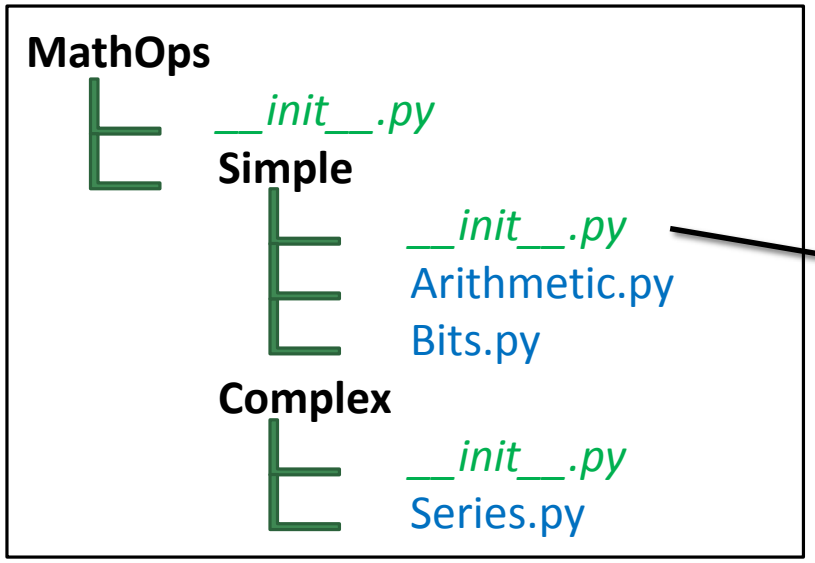
```
from MathOps.Simple import *

print (Arithmetic.Add(2,3))
print (Bits.SHL(2,3))
```

Output

```
Package MathOps init
Package MathOps.Simple init
Traceback (most recent call last):
  File “test.py", line 3, in <module>
    print (Arithmetic.Add(2,3))
NameError: name 'Arithmetic' is not defined
```

Packages

To be able to use a syntax similar to "from <module> import *" a module variable "*__all__*" must be defined. That variable will hold a list of all modules that belongs to that package.



Python 2.x / 3.x
File: __init__.py

```
print ("Package MathOps.Simple init
        with __all__ set")
__all__ = ["Arithmetic","Bits"]
```

Packages

Usage:

Python 2.x / 3.x

```
from MathOps.Simple import *

print (Arithmetic.Add(2,3))
print (Bits.SHL(2,3))
```

Output

```
Package MathOps init
Package MathOps.Simple init with __all__ set
5
16
```

Modules/Packages

If you want a module and/or package to be available to all the scripts that are executed on that system just copy the module or the entire package folder on the Python search path and you will be able to access it directly. These paths are:

- **Windows:** <PythonFolder>\Lib (Exemple: C:\Python27\Lib or C:\Python35\Lib)
- **Linux:** /usr/lib/<PythonVersion> (Example: /usr/lib/python2.7 or /usr/lib/python3.5)

Python 2.x
File: C:\Python27\Lib\MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x
File: test.py

```
import MyModule
print (MyModule.Sum(10,20))
import MyModule
```

Output

```
MyModule loaded
30
```

Modules/Packages

Python also has a special library (importlib) that can be use to dynamically import a module.

- ***importlib.****import_module* (moduleName,package=None) ➔ to import a module
- ***importlib.****reload* (module) ➔ to reload a module that was already loaded

Python 2.x
File: C:\Python27\Lib\MyModule.py

```
def Sum(x,y):
        return x+y
print ("MyModule loaded")
```

Python 2.x
File: test.py

```
import importlib

m = importlib.import_module("MyModule")
print (m.Sum(10,20))
```

Output

```
MyModule loaded
30
```

Dynamic code

Python has a keyword (exec) that can be used to dynamically compile and execute python code.

The format is exec (code, [global],[local]) where [global] and [local] represents a list of global and local definition that should be used when executing the code.

Python 2.x / 3.x

```
exec("x=100")
print(x)
```

Output

100

```
exec("def num_sum(x,y): return x+y")
print(num_sum(10,20))
```

Output

30

```
s = "abcdefg"
exec("s2=s.upper()")
print(s2)
```

Output

ABCDEFG

Dynamic code

Because of this keyword, python can obfuscate or modify itself during runtime.

Python 2.x / 3.x

```
data = [0x65, 0x66, 0x67, 0x21, 0x54, 0x76, 0x6E, 0x62, 0x29, 0x79,
        0x2D, 0x7A, 0x2D, 0x7B, 0x2A, 0x3B, 0x0E, 0x0B, 0x0A, 0x73,
        0x66, 0x75, 0x76, 0x73, 0x6F, 0x21, 0x79, 0x2C, 0x7A, 0x2C,
        0x7B]
s = ""
for i in data:
        s += chr(i-1)
exec(s)
print(Suma(1,2,3))
```

Output

6

Dynamic code

Because of this keyword, python can obfuscate or modify itself during runtime.

Python 2.x / 3.x

```
data = [0x65, 0x66, 0x67, 0x21, 0x54, 0x76, 0x6E, 0x62, 0x29, 0x79,
        0x2D, 0x7A, 0x2D, 0x7B, 0x2A, 0x3B, 0x0E, 0x0B, 0x0A, 0x73,
        0x66, 0x75, 0x76, 0x73, 0x6F, 0x21, 0x79, 0x2C, 0x7A, 0x2C,
        0x7B]
s = ""
for i in data:
        s += chr(i-1)
exec(s)
print(Suma(1,2,3))
```

```
def Suma(x,y,z):
    return x+y+z
```

Output

6

Programming in Python

GAVRILUT DRAGOS

COURSE 6

Regular expressions support

Regular expression are implemented in Python in library “re”.

Usual usage:

- regular expression (string form) is compiled into an binary form (usually an automata)
- The binary form is used for the following:
  - Checks if a string matches a regular expression
  - Checks if a sub-string from a string can be identified using a regular expression
  - Replace substrings from a string based on a regular expression

Details about re module in Python:

- Python 2: https://docs.python.org/2/library/re.html
- Python 3: https://docs.python.org/3/library/re.html

Regular expressions support

Regular expression special characters (here is a simple set of special characters)

Character	Match	Character	Match
.	All characters except new line	\d	Decimal characters 0,1,2,3,…9
^	Matches at the start of the string	\D	All except decimal characters
$	Matches at the end of the string	\s	Space, tab, new line (CR/LF) characters
*	>=0 repetition(s)	\S	All except characters designated by \s
?	0 or 1 occurrence	\w	Word characters a-z, A-Z, 0-9 and _
+	>=1 repetition(s)	\W	All except characters designated by \w
{x}	Matches <x> times	\	Escape character
{x,y}	Matches between <x> and <y> times	[^…]	Not specified group of characters
[]	Group of characters	(…)	Grouping
|	Or condition	[..-..]	'-' interval for a group of characters.

Regular expressions support

Usage:

- use re.compile (regular_expression_string,flags) to compile a regular expression into its binary form
- Use the “match” method of the resulted object to check if a string matches the regular expression

Python 2.x / 3.x

```
import re

r = re.compile("07[0-9]{8}")
if r.match("0740123456"):
        print("Match")
```

Output

```
Match
```

- The same result can be achieved by using the “match” function from the re module directly

Python 2.x / 3.x

```
import re
if re.match("07[0-9]{8}", "0740123456"):
        print("Match")
```

Regular expressions support

Pattern	String that will be match
\w+\s+\w+	“Gavrilut Dragos”, “Gavrilut Dragos Teodor”
^\w+\s+\w+$	“Gavrilut Dragos”
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}	“192.168.0.1”, “999.999.999.999”
([0-9]{1,3}\.){3}[0-9]{1,3}	“192.168.0.1”, “999.999.999.999”
^((([0-9])|([1-9][0-9])|(1[0-9]{2})|(2[0-4][0-9])|(25[0-5]))\.){3}(([0-9])|([1-9][0-9])|(1[0-9]{2})|(2[0-4][0-9])|(25[0-5]))$	Will only match IP addresses
[12]\d{12}	CNP (but will not validate the correctness of the birth date
0x[0-9a-fA-F]+	A hex number
(if|then|else|while|continue|break)	A special keyword

Regular expressions support

re.match starts the matching from the beginning of the string and stops once the matching ends and not when the string ends except for the case where regular expression pattern is using the "$" character:

Python 2.x / 3.x

```
import re

if re.match("\d+","123 USD"):
        print ("Match")

if re.match("\d+","Price is 123 USD"):
        print ("Match")

if not re.match("\d+$","123 USD"):
        print ("NO Match")
```

Output

```
Match
NO Match
```

Regular expressions support

If you want to check if a regular expression pattern is matching a part of a string, the “search” method can be used:

Python 2.x / 3.x

```
import re

if re.search("\d+","Price is 123 USD"):
        print ("Found")
```

Output

```
Found
```

The same can be achived using a compiled object:

Python 2.x / 3.x

```
import re
r = re.compile(“\d+")
if r.search("Price is 123 USD"):
        print ("Found")
```

Regular expressions support

search method stops after the first match is achived.

The object returned by the **search** or **match** method is called a match object. A match object is always evaluated to **true**. If the search does not find any match, None is returned and will be evaluated to false. A match object has several members:

- **group**(index) → returns the substring that matches that specific group. If *index* is 0, the substring refers to the entire string that was matched by the regular expression
- **lastindex** → returns the index of the last object that was matched by the regular expression. To create a group within the regular expression, one must use (…).

Python 2.x / 3.x

```
import re

result = re.search("\d+","Price is 123 USD")
if result:
        print (result.group(0))
```

Output

```
123
```

Regular expressions support

In case of some operators (like ***** or **+**) they can be preceded by **?.** This will specify a NON-greedy behavior.

Python 2.x / 3.x

```
import re

result = re.search(".*(\d+)",  "File size if 12345 bytes")
if result:
        print (result.group(1))

result = re.search(".*?(\d+)", "File size if 12345 bytes")
if result:
        print (result.group(1))
```

Output

```
5
12345
```

Regular expressions support

In case of some operators (like ***** or **+**) they can be preceded by **?.** This will specify a NON-greedy behavior.

Python 2.x / 3.x

```
import re

result = re.search(".*(\d+)",  "File size if 12345 bytes")
if result:
        print (result.group(1))

result = re.search(".*?(\d+)", "File size if 12345 bytes")
if result:
        print (result.group(1))
```

Output

```
5
12345
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("(\d+)[^\d]*(\d+)","Price is 123 USD aprox 110 EUR")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,”=>”,result.group(i))
```

Output

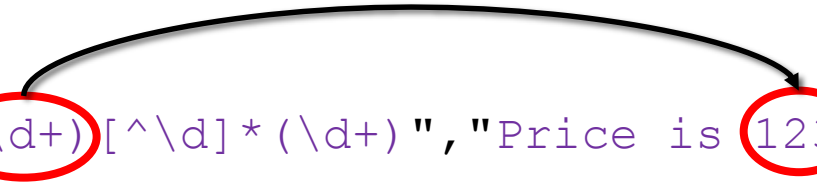
```
2
0 => 123 USD, aprox. 110
1 => 123
2 => 110
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("(\d+)[^\d]*(\d+)","Price is 123 USD aprox 110 EUR")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,”=>”,result.group(i))
```



Output

```
2
0 => 123 USD, aprox. 110
1 => 123
2 => 110
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("(\d+)[^\d]*(\d+)","Price is 123 USD aprox 110 EUR")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,"=>",result.group(i))
```

Output

```
2
0 => 123 USD, aprox. 110
1 => 123
2 => 110
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("((\d+),(\d+))[^\d]*(\d+)",
                   "Color from pixel 20,30 is 123")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,"=>",result.group(i))
```

Output

```
4
0 => 20,30 is 123
1 => 20,30
2 => 20
3 => 30
4 => 123
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("((\d+),(\d+))[^\d]*(\d+)",
                   "Color from pixel 20,30 is 123")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,"=>",result.group(i))
```

Output

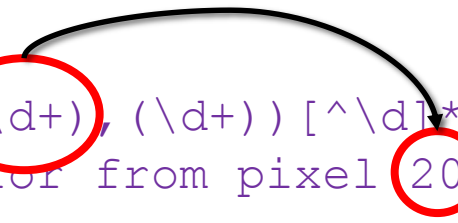
```
4
0 => 20,30 is 123
1 => 20,30
2 => 20
3 => 30
4 => 123
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("((\d+),(\d+))[^\d]*(\d+)",
                   "Color from pixel 20 30 is 123")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,"=>",result.group(i))
```



Output

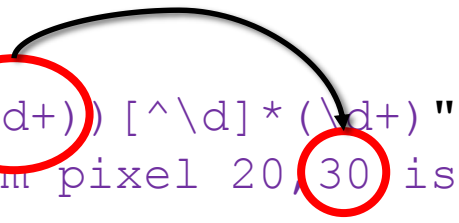
```
4
0 => 20,30 is 123
1 => 20,30
2 => 20
3 => 30
4 => 123
```

Regular expressions support

(…) is usually used to delimit specific sequences of sub-string within the regular expression pattern:

Python 2.x / 3.x

```
import re

result = re.search("((\d+),(\d+))[^\d]*(\d+)",
                   "Color from pixel 20,30 is 123")
if result:
        print (result.lastindex)
        for i in range(0,result.lastindex+1):
                print (i,”=>”,result.group(i))
```



Output

```
4
0 => 20,30 is 123
1 => 20,30
2 => 20
3 => 30
4 => 123
```

Regular expressions support

search stops after the first match. To find all substring that match a specific regular expression from a string, use **findall** method.

Python 2.x / 3.x

```
import re

result = re.findall("\d+","Color from pixel 20,30 is 123")
if result:
        print (result)
```

Output

```
['20', '30', '123']
```

The result is a vector containing all substrings that matched the regular expression.

Regular expressions support

search stops after the first match. To find all substring that match a specific regular expression from a string, use **findall** method.

Using groups (…) is also allowed (in this case they will be converted to a tuple in each list element.

Python 2.x / 3.x

```
import re

result = re.findall("(\d)(\d+)","Color from pixel 20,30 is 123")
if result:
        print (result)
```

Output

```
[('2', '0'), ('3', '0'), ('1', '23')]
```

Regular expressions support

search stops after the first match. To find all substring that match a specific regular expression from a string, use **findall** method.

Using groups (…) is also allowed (in this case they will be converted to a tuple in each list element.

Python 2.x / 3.x

```
import re

result = re.findall("(\d)(\d+)","Color from pixel 20,30 is 123")
if result:
        print (result)
```

Output

```
[('2', '0'), ('3', '0'), ('1', '23')]
```

Regular expressions support

search stops after the first match. To find all substring that match a specific regular expression from a string, use **findall** method.

Using groups (…) is also allowed (in this case they will be converted to a tuple in each list element.

Python 2.x / 3.x

```
import re

result = re.findall("(\d)(\d+)","Color from pixel 20,30 is 123")
if result:
        print (result)
```

Output

```
[('2', '0'), ('3', '0'), ('1', '23')]
```

Regular expressions support

split method can be used to split a string using a regular expression.

The result is a vector with all elements that substrings that were obtained after the split occurred.

Python 2.x / 3.x

```
import re

result = re.split("[aeiou]+","Color from pixel 20,30 is 123")
print (result)
```

Output

['C', 'l', 'r fr', 'm p', 'x', 'l 20,30 ', 's 123']

Regular expressions support

Groups can also be used. In this case the split is done after each group that matches.

Python 2.x / 3.x

```
import re
print (re.split("\d\d","Color from pixel 20,30 is 123"))
```

Elements

'Color from pixel '	','	' is '	'3'

Python 2.x / 3.x

```
import re
print (re.split("(\d)(\d)","Color from pixel 20,30 is 123"))
```

Elements

'Color from pixel '	'2'	'0'	','	'3'	'0'	' is '	'1'	'2'	'3'

Regular expressions support

Groups can also be used. In this case the split is done after each group that matches.

Python 2.x / 3.x

```
import re
print (re.split("\d\d+","Color from pixel 20,30 is 123"))
```

Elements

'Color from pixel ' ',' ' is ' ''

Python 2.x / 3.x

```
import re
print (re.split("(\d)(\d+)","Color from pixel 20,30 is 123"))
```

Elements

'Color from pixel ' '2' '0' ',' '3' '0' ' is ' '1' '23' ''

Regular expressions support

split method also allow flags and to specify how many times a split can be performed. The full format is: *split (patterm, string, maxsplit=0, flags=0)*

Python 2.x / 3.x

```
import re
s = "Today I'm having a python course"
print (re.split("[^a-z']+", s))
print (re.split("[^a-z']+", s, 2))
print (re.split("[^a-z']+", s, flags = re.IGNORECASE))
print (re.split("[^a-z']+", s, 2, flags = re.IGNORECASE))
print (re.split("[^a-z'A-Z]+", s))
```

Output

```
['', 'oday', "'m", 'having', 'a', 'python', 'course']
['', 'oday', "'m having a python course"]
['Today', "I'm", 'having', 'a', 'python', 'course']
['Today', "I'm", 'having a python course']
['Today', "I'm", 'having', 'a', 'python', 'course']
```

Regular expressions support

Regexp can also be used to replace an a match with another string using the method **sub**.
format is: *sub (pattern, replace, string, count=0, flags=0)*

- ***pattern*** is a regular expression to search for
- ***replace*** is either a string or a function
- ***string*** is the string where you are going to search the pattern
- ***count*** represents how many time the replacement can occur. If missing or 0 means for all matches.
- ***flags*** represents some flags (like re.IGNORECASE)

Python 2.x / 3.x

```
import re
s = "Today I'm having a python course"
print (re.sub("having\s+a\s+\w+\s+course", "not doing anything", s))
```

Output

Today I'm **not doing anything**

Regular expressions support

Regexp can also be used to replace an a match with another string using the method **sub**.
format is: *sub (pattern, replace, string, count=0, flags=0)*

If ***replace*** parameter is a string there is a special operator (**\<number>**) that if found within the replacement string will be replace with the group from the match search (for example \3 will be replaced with .group(3)).

Python 2.x / 3.x

```
import re
s = "Today I'm having a python course"
print (re.sub("having\s+a\s+(\w+)\s+course",
              r"not doing the \1 course",
               s))
```

Output

Today I'm **not doing the python course**

Regular expressions support

Regexp can also be used to replace an a match with another string using the method **sub**.
format is: *sub (pattern, replace, string, count=0, flags=0)*

If ***replace*** parameter is a function there is a special operator (**\<number>**) that if found within the replacement string will be replace with the group from the match search (for example \3 will be replaced with .group(3)).

Python 2.x / 3.x

```
import re
s = "Today I'm having a python course"
print (re.sub("having\s+a\s+(\w+)\s+course",

              r"not doing the \1 course",
               s))
```

You can also use \g<number> with the same effect. In this case **\g<1>**

Regular expressions support

Regexp can also be used to replace an a match with another string using the method **sub**.
format is: *sub (pattern, replace, string, count=0, flags=0)*

If ***replace*** parameter is a function it receives the match object. Usually that function will use .group(0) method to get the string that was matched and convert it to the replacement value.

Python 2.x / 3.x

```
import re

def ConvertToHex(s):
        return hex(int(s.group(0)))

s = "File size is 12345 bytes"
print (re.sub("\d+",ConvertToHex, s))
```

Output

File size is 0x3039 bytes

Extensions

Python regular expressions supports extensions. The form of the extension is **(?...)**

- **(?P<name>…)** will set the name of a group to a given string. In case of a match, that group can be accessed based on it's name.

Python 2.x / 3.x

```
import re
s = "File size if 12345 bytes"
result = re.search("(?P<file_size>\d+)",s)
if result:
        print ("Size is ",result.group("file_size"))
```

Output

```
Size is 12345
```

Extensions

Python regular expressions supports extensions. The form of the extension is **(?...)**

- **(?P<name>…)** →The match object also has a **groupdict** method that returns a dictionary with all the keys and strings that match the specified regular expression

Python 2.x / 3.x

```
import re

s = "File config.txt was create on 2016-10-20 and has 12345 bytes"
result = re.search("File\s+(?P<name>[a-z\.]+)\s.*(?P<date>\d{4}-\d{2}-
\d{2}).*\s(?P<size>\d+)",s)
if result:
        print (result.groupdict())
```

Result

```
{
      'date'   : '2016-10-20',
      'name'   : 'config.txt',
      'size'   : '12345‘
}
```

Extensions

Python regular expressions supports extensions. The form of the extension is **(?...)**

- **(?i)(…)** ignore case will be applied for the current block match
- **(?s)(…)** "." (dot) will match everything

Python 2.x / 3.x

```
import re
s = "12345abc54321"
result = re.search("(?i)([A-Z]+))",s)
if result:
        print (result.group(1))
```

Output

```
abc
```

Extensions

Python regular expressions supports extensions. The form of the extension is **(?...)**

- **(?=…)** will match the previous expression only if next expression is … (this is called look ahead assertion)
- **(?!…)** similarly, will match only if the next expression will **not** match …

Python 2.x / 3.x

```
import re
s = "Python Course"
result = re.search("(Python)\s+(?=Course)",s)
if result:
        print (result.group(1))
```

Output

```
Python
```

Extensions

Python regular expressions supports extensions. The form of the extension is **(?...)**

- **(?#...)** represents a comment / information that can be added in the regular expression to reflect the purpose of a specific group

Python 2.x / 3.x

```
import re
s = "Size is 1234 bytes"
result = re.search("(?# file size)(\d+)",s)
if result:
        print ("Size is ",result.group(1))
```

Output

```
Size is 1234
```

Building a tokenizer

Python has a way to iterate through a string applying different regular expression. Because of this, a tokenizer can be built for different languages. Use method **finditer** for this.

Python 2.x / 3.x

```
import re
number = "(?P<number>\d+)"
operation = "(?P<operation>[+\-\*\/])"
braket = "(?P<braket>[\(\)])"
space = "(?P<space>\s)"
other = "(?P<other>.)"
r = re.compile(number+"|"+operation+"|"+braket+
               "|"+space+"|"+other)
expr = "10 * (250+3)"
for matchobj in r.finditer(expr):
       key = matchobj.lastgroup
       print (matchobj.group(key)+" => "+key)
```

Output

```
10 => number
  => space
* => operation
  => space
( => braket
250 => number
+ => operation
3 => number
) => braket
```

Regular expressions support

Recommendations:

1. If the same regular expression is used multiple times using it in the compile form will improve the performance of the script
2. Even if Python recognizes some escape sequences (such as \d or \w) it is better to either use a raw string r"…" or to duplicate the escape character
   - Instead of "\d" → use r"\d" or "\\d"
3. Regular expression need memory. If all you need is to search a substring within another substring or perform simple string operation, don't use regular expression for this.
4. If you are trying to use the regular expression in a portable way, don't use some features like (?P=name) → other languages or regular expression engines might not support this.

Programming in Python

GAVRILUT DRAGOS

COURSE 7

Time module

Implements function that allows one to work with time:

- Get current time
- Format time
- Sleep
- Time zone information

Details about **time** module in Python:

- Python 2: https://docs.python.org/2/library/time.html#module-time
- Python 3: https://docs.python.org/3/library/time.html#module-time

Time module

Usage:

Python 2.x / 3.x

```
import time
w = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
print ("Time in seconds:",time.time())
print ("Today           :",time.ctime())
tmobj = time.localtime()
print ("Year            :",tmobj.tm_year)
print ("Month           :",tmobj.tm_mon)
print ("Day             :",tmobj.tm_mday)
print ("Day of week     :",w[tmobj.tm_wday])
print ("Day from year   :",tmobj.tm_yday)
print ("Hour            :",tmobj.tm_hour)
print ("Min             :",tmobj.tm_min)
print ("Sec             :",tmobj.tm_sec)
```

Output

```
Time in seconds: 1478936424.0649655
Today     : Sat Nov 12 09:40:24 2016
Year            : 2016
Month           : 11
Day             : 12
Day of week     : Saturday
Day from year   : 317
Hour            : 9
Min             : 40
Sec             : 24
```

Time module

Both **localtime** and **gmtime** have one parameter (the number of seconds from 1970). If this parameter is provided the time object will be the time computed based on that number. Otherwise the time object will be the time based on time.time () (current time) value.

Python 2.x / 3.x

```
import time

print (time.localtime())
print (time.gmtime())
print (time.gmtime(100))
print (time.gmtime(time.time()))
```

Output

```
time.struct_time(tm_year=2016, tm_mon=11, tm_mday=12,
                 tm_hour=9, tm_min=53, tm_sec=47,
                 tm_wday=5, tm_yday=317, tm_isdst=0)
time.struct_time(tm_year=2016, tm_mon=11, tm_mday=12,
                 tm_hour=7, tm_min=53, tm_sec=47,
                 tm_wday=5, tm_yday=317, tm_isdst=0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1,
                 tm_hour=0, tm_min=1, tm_sec=40,
                 tm_wday=3, tm_yday=1, tm_isdst=0)
time.struct_time(tm_year=2016, tm_mon=11, tm_mday=12,
                 tm_hour=7, tm_min=53, tm_sec=47,
                 tm_wday=5, tm_yday=317, tm_isdst=0)
```

Time module

Use **mktime** to convert from a time object struct to a float number.
Use **asctime** member to convert from a time object to a readable representation of the time (string format).

Python 2.x / 3.x

```
import time

t = time.time()
tobj = time.localtime()
tm = time.mktime(tobj)
print (tm)
print (t)
print (time.asctime(tobj))
```

Output

```
1478938309.0
1478938309.444829
Sat Nov 12 10:11:49 2016
```

Time module

Use **strftime** to time object to a specified string representation:

Abreviation	Description
%H	Hour in 24 hour format
%I	Hour in 12 hour format
%Y	Year (4 digits)
%m	Month (decimal)
%B	Month (name)

Abreviation	Description
%M	Minute
%S	Seconds
%A	Day of week (name)
%d	Day of month (decimal)
%p	AM or PM

Python 2.x / 3.x

```
import time
tobj = time.localtime()
print (time.strftime("%H:%M:%S - %Y-%m-%d",tobj))
print (time.strftime("%I%p:%M:%S - %B",tobj))
print (time.strftime("%B,%A %d %Y",tobj))
```

Output

```
10:25:20 - 2016-11-12
10AM:25:20 - November
November,Saturday 12 2016
```

Time module

strftime if used without a time object applies the string format to the current time.

Time module also has a function **sleep** that receives one parameter (the number of seconds the current script has to wait until it continues its execution).

Python 2.x / 3.x

```
import time

for i in range(0,8):
        print (time.strftime("%H:%M:%S"))
        time.sleep(2)#sleep 2 seconds
```

If no time object structure is provided, **strftime** function will use current time to apply the specified format.

Output

```
11:46:15
11:46:17
11:46:19
11:46:21
11:46:23
11:46:25
11:46:27
11:46:29
```

hashlib module

Implements function that allows one to compute different cryptographic functions:

- MD5
- SHA-1
- SHA-224
- SHA-384
- SHA-512

Details about **hashlib** module in Python:

- Python 2: https://docs.python.org/2/library/hashlib.html
- Python 3: https://docs.python.org/3/library/hashlib.html

hashlib module

Each hashlib object has an **update** function (to update the value of the hash) and a **digest** or **hexdigest** function(s) to compute the final hash.

Python 2.x / 3.x

```
import hashlib

m = hashlib.md5()
m.update(b"Today")
m.update(b" I'm having")
m.update(b" a Python ")
m.update(b"course")
print (m.hexdigest())
```

Output

```
9dea650a4eab481ec0f4b5ba28e3e0b8
```

```
import hashlib

print (hashlib.md5(b"Today I'm having a Python course").hexdigest())
```

hashlib module

Each hashlib object has an **update** function (to update the value of the hash) and a **digest** or **hexdigest** function(s) to compute the final hash.

Python 2.x / 3.x

```
import hashlib

m = hashlib.md5()
m.update(b"Today")
m.update(b" I'm having")
m.update(b" a Python ")
m.update(b"course")
print (m.hexdigest())
```

The **<b>** prefix in front of a string is ignored in Python 2. In Python 3 means that the string is a byte list. **update** method requires a list of bytes (not a string). However, in Python 2 it can be used without the prefix **<b>**

```
import hashlib

print (hashlib.md5(b"Today I'm having a Python course").hexdigest())
```

hashlib module

Hashes are often use on files (to associate the content of a file to a specific hash).

Python 2.x / 3.x

```
import hashlib

def GetFileSHA1(filePath):
    m = hashlib.sha1()
    m.update(open(filePath,"rb").read())
    return m.hexdigest()


print (GetFileSHA1("< a file path >"))
```

Output

```
cad7a796be26149218a76661d316685d7de2d56d
```

While this example is ok, keep in mind that it loads the entire file content in memory !!!

hashlib module

The correct way to do this (having a support for large files is as follows):

Python 2.x / 3.x

```
import hashlib
def GetFileSHA1(filePath):
    try:
        m = hashlib.sha1()
        f = open(filePath,"rb")
        while True:
            data = f.read(4096)
            if len(data)==0: break
            m.update(data)
        f.close()
        return m.hexdigest()
    except:
        return ""
```

JSON module

Python has several implementations for data serialization.

- JSON
- Pickle
- Marshal

Documentation for JSON:

- Python 2: https://docs.python.org/2/library/json.html
- Python 3: https://docs.python.org/3/library/json.html

JSON module

JSON functions:

- **json.dump** (obj, fp, *skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw*)
- **json.dumps**(obj, *skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw*) ➔ to obtain the string representation of the obj in JSON format
- **json.load**(fp, *cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw*)
- **json.loads**(s, *encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw*)

JSON module

Usage (serialization):

Python 2.x / 3.x

```
import json

d = { "a":[1,2,3],
      "b":100,
      "c":True
}
s = json.dumps(d)
open("serialization.json","wt").write(s)
print (s)
```

serialization.json

```
FileAddr     000 001 002 003 004 005 006 007     Text
000000000    123 034 099 034|058 032 116 114     {"c": tr
000000008    117 101 044 032|034 097 034 058     ue, "a":
000000016    032 091 049 044|032 050 044 032      [1, 2,
000000024    051 093 044 032|034 098 034 058     3], "b":
000000032    032 049 048 048|125                  100}
```

Output

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

JSON module

Usage (de-serialization):

Python 2.x / 3.x

```
import json

data = open("serialization.json","rt").read()
d = json.loads(data)
print (d)
```

```
import json

d = json.load(open("serialization.json","rt"))
print (d)
```

Output

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

PICKLE module

Pickle is another way to serialize objects in Python. The serialization is done in a binary mode.

Pickle can also serialize:

- Functions (defined using **def** and not lambda)
- classes
- Functions from modules

Documentation for PICKLE :

- Python 2: https://docs.python.org/2/library/pickle.html
- Python 3: https://docs.python.org/3/library/pickle.html

PICKLE module

PICKLE functions:

- **pickle.dump** (obj, file, *protocol=None, *, fix_imports=True*)
- **pickle.dumps**(obj, *protocol=None, *, fix_imports=True*) ➔ to obtain the buffer representation of the obj in pickle format
- **pickle.load**(file, **, fix_imports=True, encoding="ASCII", errors="strict"*)
- **pickle.loads**(byte_object, **, fix_imports=True, encoding="ASCII", errors="strict"*)

PICKLE support multiple version. Be careful when you serialize with Python 2 and try to de-serialize with Python 3 (not all version supported by Python 3 are also supported by Python 2).

If you are planning to switch between versions, either check pickle.HIGHEST_PROTOCOL to see if the hightest protocol are compatible, or use **0** as the protocol value.

PICKLE module

Usage (serialization):

Python 2.x / 3.x

```
import pickle

d = {
    "a":[1,2,3],
    "b":100,
    "c":True
}

buffer = pickle.dumps(d) #buffer = pickle.dumps(d,0) (safety)
open("serialization.pickle","wb").write(buffer)
```

serialization.json

FileAddr	000	001	002	003	004	005	006	007	Text
000000000	128	003	125	113	000	040	088	001	Ç♥}q (X☺
000000008	000	000	000	099	113	001	136	088	cqêX
000000016	001	000	000	000	097	113	002	093	☺ aq☻]
000000024	113	003	040	075	001	075	002	075	q♥(K☺K☻K
000000032	003	101	088	001	000	000	000	098	♥eX☺ b
000000040	113	004	075	100	117	046			q♦Kdu.

Pickle need a file to be open in binary mode !

PICKLE module

Usage (de-serialization):

Python 2.x / 3.x

```
import pickle

data = open("serialization.pickle","rb").read()
d = pickle.loads(data)
print (d)
```

```
import pickle

d = pickle.load(open("serialization.pickle","rb"))
print (d)
```

Output

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

Marshal module

Marshal is another way to serialize objects in Python. The serialization is done in a binary mode. Designed for python compiled code (pyc). The binary result is platform-dependent !!!

Marshal functions:

- **marshal.dump** (value, file, *[version]*) marshal
- **marshal.dumps(**value, *[version]*) ➔ to obtain the binary representation of the obj in marshal format
- **marshal.load(**file)
- **marshal.loads(**string/buffer)

Documentation for Marshal :

- Python 2: https://docs.python.org/2/library/marshal.html#module-marshal
- Python 3: https://docs.python.org/3/library/marshal.html#module-marshal

Marshal module

Usage (serialization):

Python 2.x / 3.x

```
import marshal

d = {
    "a":[1,2,3],
    "b":100,
    "c":True
}

buffer = marshal.dumps(d)
open("serialization.marshal","wb").write(buffer)
```

serialization.json

FileAddr	000	001	002	003	004	005	006	007	Text
000000000	251	218	001	099	084	218	001	097	√┌☺cT┌☺a
000000008	091	003	000	000	000	233	001	000	[♥ Θ☺
000000016	000	000	233	002	000	000	000	233	ΘΘ Θ
000000024	003	000	000	000	218	001	098	233	♥ ┌☺bΘ
000000032	100	000	000	000	048				d 0

Marshal module

Usage (de-serialization):

Python 2.x / 3.x

```
import marshal

data = open("serialization.marshal","rb").read()
d = marshal.loads(data)
print (d)
```

```
import marshal

d = marshal.load(open("serialization.marshal","rb"))
print (d)
```

Marshal serialization has a different format in Python 2 and Python 3 (these two are not compatible).

Output

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

Random module

Implements different random base functions:

- random.random() ➔ a random float number between 0 and 1
- random.randint(min,max) ➔ a random integer number between [min … max]
- random.choice(list) ➔ selects a random element from a list
- random.shuffle(list) ➔ shuffles the list
- random.sample(list,count) ➔ creates another list from the current one containing **count** elements

Details about **random** module in Python:

- Python 2: https://docs.python.org/2/library/random.html
- Python 3: https://docs.python.org/3/library/random.html

Random module

Usage:

Python 2.x / 3.x

```
import random

print (random.random())
print (random.randint(5,10))

l = [2,3,5,7,11,13,17,19]
print (random.choice(l))
print (random.sample(l,3))

random.shuffle(l)
print (l)
```

Output

```
0.9410874890940395
9
5
[19, 17, 11]
[13, 17, 11, 5, 2, 19, 7, 3]
```

ZipFile module

Implements different functions to work with a zip archive:

- List all elements from a zip archive
- Extract files
- Add files to archive
- Get file information
- etc

Details about **zipfile** module in Python:

- Python 2: https://docs.python.org/2/library/zipfile.html
- Python 3: https://docs.python.org/3/library/zipfile.html

ZipFile module

Listing the content of a zip archive:

Python 2.x / 3.x

```
import zipfile

z = zipfile.ZipFile("archive.zip")
for i in z.infolist():
        print (i.filename,
               i.file_size,
               i.compress_size)
z.close()
```

Output

```
MathOps/ 0 0
MathOps/Complex/ 0 0
MathOps/Complex/Series.py 117 79
MathOps/Complex/__init__.py 38 38
MathOps/Simple/ 0 0
MathOps/Simple/Arithmetic.py 54 52
MathOps/Simple/Bits.py 60 55
MathOps/Simple/__init__.py 87 84
MathOps/__init__.py 30 30
a.py 43 43
all.csv 62330588 8176706
```

ZipFile module

To extract a file from an archive:

Python 2.x / 3.x

```
import zipfile
z = zipfile.ZipFile("archive.zip")
z.extract("MathOps/Simple/Arithmetic.py","MyFolder")
z.close()
```

Arithmetic.py will be extracted to “MyFolder/MathOps/Simple/Arithmetic.py”

To extract all files:

Python 2.x / 3.x

```
import zipfile
z = zipfile.ZipFile("archive.zip")
z.extractall("MyFolder")
z.close()
```

ZipFile module

A file can also be opened directly from an archive. This is usually required if one wants to extract the content somewhere else or if the content needs to be analyzed in memory.

Python 2.x / 3.x

```
import zipfile

z = zipfile.ZipFile("archive.zip")
f = z.open("MathOps/Simple/Arithmetic.py")
data = f.read()
f.close()
open("my_ar.py","wb").write(data)
z.close()
```

Method **open** from zipfile returns a file-like object. You can also specify a password:
Format: ZipFile.*open*(name, *mode='r', pwd=None*)

ZipFile module

The following script creates a zip archive and add files to it:

Python 2.x / 3.x

```
import zipfile

z = zipfile.ZipFile("new_archive.zip","w",zipfile.ZIP_DEFLATED)
z.writestr("test.txt","some texts ...")
z.write("serialization.json")
z.write("serialization.json", "/dir/a.json")
z.writestr("/dir/a.txt","another text ...")
z.close()
```

writestr method writes the content of a string into a zip file.
write methods add a file to the archive.

When creating an archive one can specify a desire compression: ZIP_DEFLATE, ZIP_STORED, ZIP_BZIP2 or ZIP_LZMA.

Programming in Python

GAVRILUT DRAGOS

COURSE 8

Sockets

Sockets are implemented through **socket** module in Python.

A socket object in Python has the same functions as a normal socket from C/C++: accept, bind, close, connect, listen, recv, send, …

Besides this several other functions are available for domain translation, time outs, etc

Documentation for Python socket module can be found on:

- Python 2: https://docs.python.org/2/library/socket.html
- Python 3: https://docs.python.org/3/library/socket.html

Sockets

How to build a simple server/client in Python:

Python 2.x/3.x (Server)

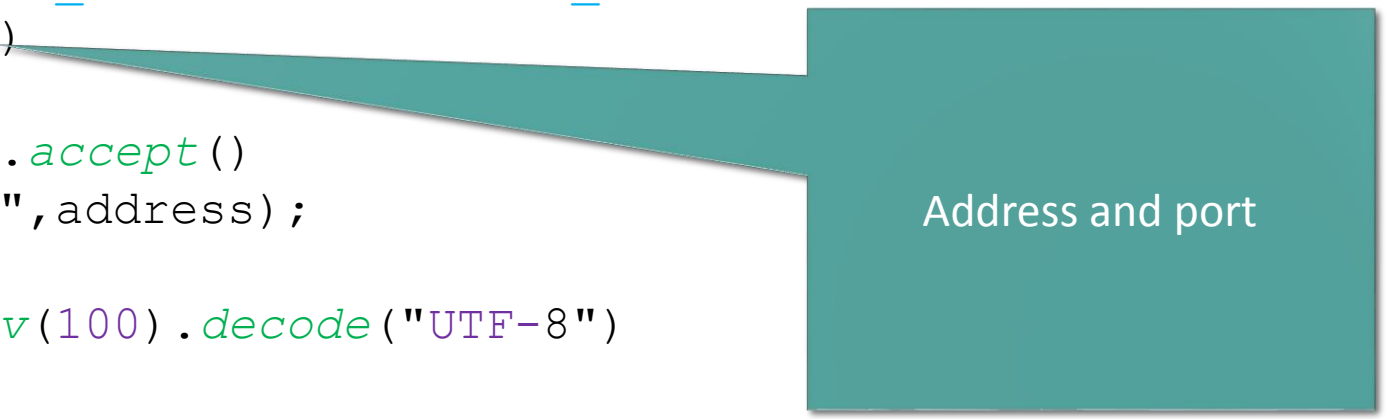
```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1",1234))
s.listen(1)
(connection, address) = s.accept()
print ("Connectd address:",address);
while True:
    data = connection.recv(100).decode("UTF-8")
    if not data: break
    print("Received: ",data)
    if "exit" in data: break
connection.close()
print ("Server closed")
```

Sockets

How to build a simple server/client in Python:

Python 2.x/3.x (Server)

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1",1234))
s.listen(1)
(connection, address) = s.accept()
print ("Connectd address:",address);
while True:
    data = connection.recv(100).decode("UTF-8")
    if not data: break
    print("Received: ",data)
    if "exit" in data: break
connection.close()
print ("Server closed")
```

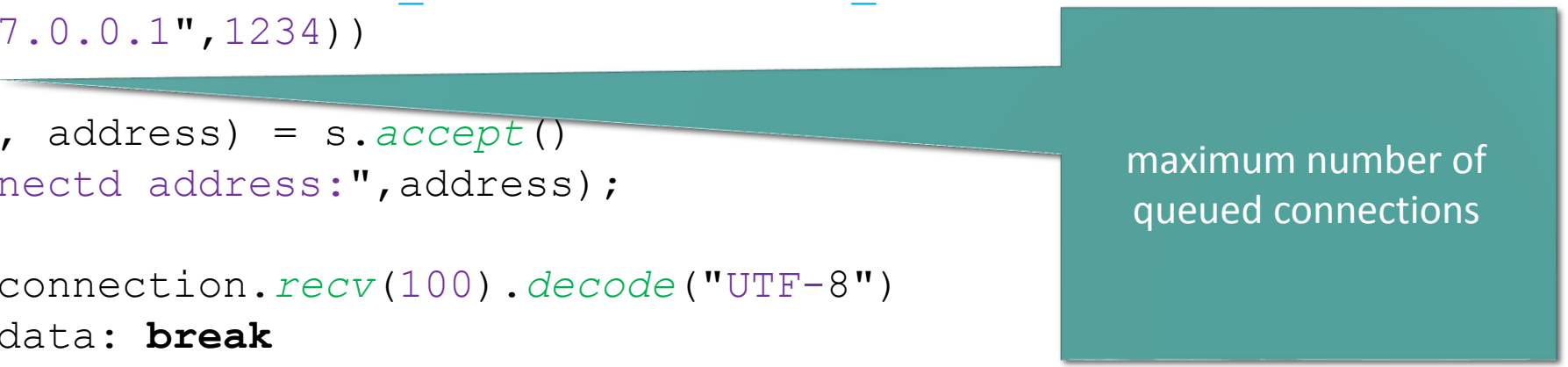


Sockets

How to build a simple server/client in Python:

Python 2.x/3.x (Server)

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1",1234))
s.listen(1)
(connection, address) = s.accept()
print ("Connectd address:",address);
while True:
    data = connection.recv(100).decode("UTF-8")
    if not data: break
    print("Received: ",data)
    if "exit" in data: break
connection.close()
print ("Server closed")
```

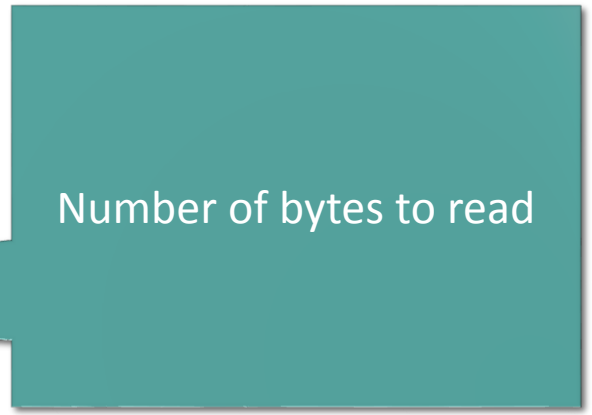


Sockets

How to build a simple server/client in Python:

Python 2.x/3.x (Server)

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1",1234))
s.listen(1)
(connection, address) = s.accept()
print ("Connectd address:",address);
while True:
    data = connection.recv(100).decode('utf-8')
    if not data: break
    print("Received: ",data)
    if "exit" in data: break
connection.close()
print ("Server closed")
```



Sockets

How to build a simple server/client in Python:

Python 2.x/3.x (Server)

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1",1234))
s.listen(1)
(connection, address) = s.accept()
print ("Connectd address:",address);
while True:
    data = connection.recv(100).decode("UTF-8"
    if not data: break
    print("Received: ",data)
    if "exit" in data: break
connection.close()
print ("Server closed")
```

Use .decode("UTF-8") to convert a byte array to a string. Encoding in this case is done with UTF-8

Sockets

How to build a simple server/client in Python:

Python 2.x/3.x (Client)

```
import socket,time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1",1234))
s.send(b"Mesaj 1")
time.sleep(1)
s.send(b"Mesaj 2")
time.sleep(1)
s.send(b"exit")
s.close()
```

Output from the server

```
Connectd address: ('127.0.0.1', 61266)
Received:  Mesaj 1
Received:  Mesaj 2
Received:  exit
Server closed
```

Sockets

Getting current system IP

Python 2.x/3.x

```
import socket
print (socket.gethostbyname(socket.gethostname()))
```

Convert a host to an IP:

Python 2.x/3.x

```
import socket
print (socket.gethostbyname('uaic.ro'))
```

Getting the name associated with an IP:

Python 2.x/3.x

```
import socket
print (socket.gethostbyaddr("85.122.16.7"))
```

Output

```
('jad.uaic.ro', [],
['85.122.16.7'])
```

Sockets

Checking if a port is open:

Python 2.x/3.x (Client)

```
import socket

ip = "127.0.0.1"
ports = [20, 21, 23, 25, 80, 443, 530, 8080]
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(3) #3 seconds timeout
for port in ports:
        if s.connect_ex((ip, port)) == 0:
                print("Port ",port," is open")
        else:
                print("Port ",port," is closed")
```

connect_ex returns an error code if the connection is not possible. **0** means no error.

URL modules

Python has several implementation for accessing the content of a URL. Some libraries are available only for Python 2.

The most widely used modules are urllib and urllib2 (available only for Python 2).

- Python 2: https://docs.python.org/2/library/urllib2.html
- Python 2: https://docs.python.org/2/library/urllib.html
- Python 3: https://docs.python.org/3/library/urllib.html#module-urllib

urllib2 module

Python 2 has a very simple way of downloading the content of an URL. The following script lists all professors from our faculty that are currently teaching.

Python 2.x

```
import urllib2,re

urlInfo = "http://profs.info.uaic.ro/~orar/orar_profesori.html"
r = re.compile("<li><a\shref=\"[^\"\.]*\.html\">.*([A-Za-z\.\s,]+)<\/a>")

try:
	response = urllib2.urlopen(urlInfo).read()
	for mo in r.finditer(response):
		print (mo.group(1).strip())
except Exception as e:
	print ("Error -> ",e)
```

urllib module

Python 3 has another module (urllib). A similar module exists in Python 2. The following code extracts the name of the dean of our faculty.

Python 3.x

```
import urllib,re
from urllib import request
urlManagement = 'http://www.info.uaic.ro/bin/Structure/Management'
r = re.compile(b"Dean.*<a href=\"[^\"]+\">([A-Za-z\s]+)")

try:
        response = urllib.request.urlopen(urlManagement).read()
        obj = r.search(response)
        if obj:
                print ("Our dean is : ",obj.group(1))
except Exception as e:
        print ("Error -> ",e)
```

FTP module

Python has a module (ftplib) develop to enable working with FTP servers:

- Retrieve and store files
- Enumerate files from a FTP server
- Create folder on the FTP Server
- Support for password protected servers
- Support for custom FTP commands

Documentation

- Python 2: https://docs.python.org/2/library/ftplib.html
- Python 3: https://docs.python.org/3/library/ftplib.html

FTP module

The following snippet lists all directories from ftp.debian.org from folder debian.

Python 3.x

```
from ftplib import FTP
#drwxr-sr-x   18 1176     1176         4096 Sep 17 09:55 dists
def parse_line(line):
        if line.startswith("d"):
                print (line.rsplit(" ",1)[1])
try:
        client = FTP("ftp.debian.org")
        res = client.login()
        print (res)
        client.retrlines("LIST /debian/",parse_line)
        client.quit()
except Exception as e:
        print (e)
```

Output

```
230 Login successful.
dists
doc
indices
pool
project
tools
zzz-dists
```

FTP module

The following snippet downloads a file from a FTS server.

Python 3.x

```
from ftplib import FTP

cmdToDownload = "RETR /debian/extrafiles"
try:
        client = FTP("ftp.debian.org")
        res = client.login()
        f = open("debian_extrafiles","wb")
        client.retrbinary(cmdToDownload ,lambda buf: f.write(buf))
        f.close()
        client.quit()
except Exception as e:
        print (e)
```

FTP module

List of all supported FTP commands:

Command	Description
FTP.connect	Connect to a specified host on a specified port (default is 21)
FTP.login	Specifies the user name and password for ftp login
FTP.retrlines	Send a command to the FTP server and retrieves the results. The result is send line by line to a callback function
FTP.storbinary	Stores a file in a binary mode on the FTP server
FTP.retrbinary	Retrieves a binary file from the server. A callback must be provided to write the binary content.
FTP.rename	Rename a file/folder from the server
FTP.delete	Deletes a file from the server
FTP.rmd	Deletes a folder from the server.

SMTP module

Python has a module (smtp) develop to enable working with emails.

While for simple emails (a subject and a text) the smtp module is enough, for more complex emails (attachment, etc) there is also another module (email.mime) that can be use to create an email.

Python 2 and Python 3 have some differences on how mime types can be used.

Documentation

- Python 2: https://docs.python.org/2/library/smtplib.html
- Python 2: https://docs.python.org/2/library/email.mime.html
- Python 3: https://docs.python.org/3/library/smtplib.html
- Python 3: https://docs.python.org/3/library/email.mime.html

SMTP module

The following can be used to send an email.

Python 2.x

```
import smtplib
from email.MIMEText import MIMEText
mail = smtplib.SMTP('smtp.gmail.com', 587)
mail.ehlo()
mail.starttls()
mail.login("<user_name>@gmail.com", "<Password>")
msg = MIMEText("My first email")
msg['Subject'] = "First email"
msg['From'] = "<user_name>@gmail.com"
msg['To'] = "<recipient email address>"
mail.sendmail("<from>", "<to>", msg.as_string())
mail.quit()
```

SMTP module

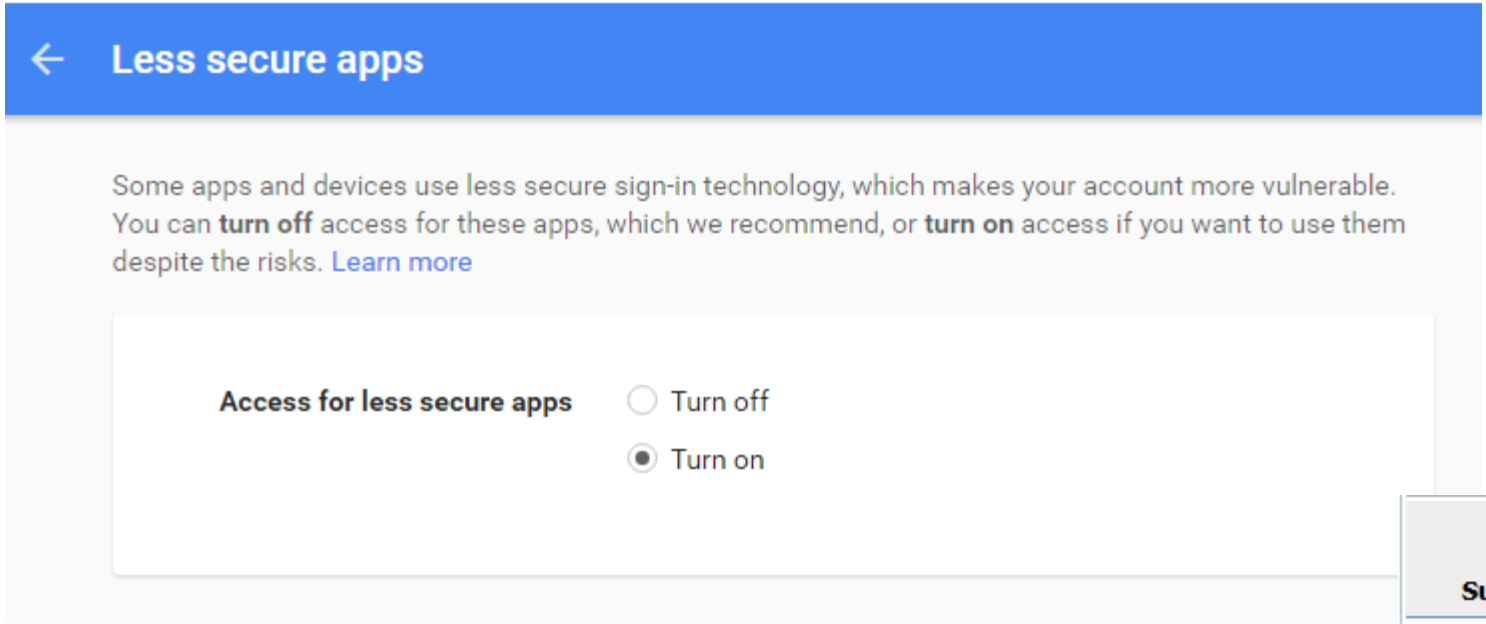
The following can be used to send an email.

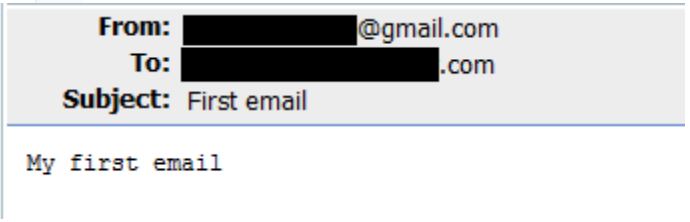
Python 3.x

```
import smtplib
from email.mime.text import MIMEText
mail = smtplib.SMTP('smtp.gmail.com', 587)
mail.ehlo()
mail.starttls()
mail.login("<user_name>@gmail.com", "<Password>")
msg = MIMEText("My first email")
msg['Subject'] = "First email"
msg['From'] = "<user_name>@gmail.com"
msg['To'] = "<recipient email address>"
mail.sendmail("<from>", "<to>", msg.as_string())
mail.quit()
```

SMTP module

For the previous snippet to work with gmail servers you need to activate “Access for less secure apps” from you google account (https://www.google.com/settings/security/lesssecureapps)





SMTP module

Sending multiple attachments.

Python 3.x

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
mail = smtplib.SMTP('smtp.gmail.com', 587)
mail.ehlo()
mail.starttls()
mail.login("<user_name>@gmail.com", "<Password>")
msg = MIMEMultipart ()
msg['Subject'] = "First email"
msg['From'] = "<user_name>@gmail.com"
msg['To'] = "<recipient email address>"
msg.attach(MIMEImage(open("image.png","rb").read()))
msg.attach(MIMEImage(open("image2.png","rb").read()))
mail.send_message(msg)
mail.quit()
```

SMTP module (attachments + body)

Python 3.x

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
from email.mime.text import MIMEText
mail = smtplib.SMTP('smtp.gmail.com', 587)
mail.ehlo()
mail.starttls()
mail.login("<user_name>@gmail.com", "<Password>")
msg = MIMEMultipart ("mixed")
msg['Subject'] = "First email"
msg['From'] = "<user_name>@gmail.com"
msg['To'] = "<recipient email address>"
msg.attach(MIMEText("The body of the email", 'plain'))
msg.attach(MIMEImage(open("image.png","rb").read()))
mail.send_message(msg)
mail.quit()
```

Building a simple HTTP server

Python has some build in modules that can simulate a http server. There are some differences between Python 2 and Python 3 in terms of module names but the basic functionality is the same.

Python 2.x

```
import SimpleHTTPServer
import SocketServer

httpd = SocketServer.TCPServer(("127.0.0.1", 9000),
                               SimpleHTTPServer.SimpleHTTPRequestHandler)
httpd.serve_forever()
```

This script will create a HTTP server that listens on port 9000.

We will discuss more about these servers after we learn about classes.

Building a simple HTTP server

The default behavior for such a server is to produce a directory listing for the root where the script is.

However, if within the root a index.html file is found, that file will be loaded and send to the client.

These modules can also be executed automatically from the command line as follows:

- Python 2: **python -m SimpleHTTPServer 9000**
- Python 3: **python -m http.server 9000**

The last parameter from the command line is the port number.