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 → 2.7.15
- 3.x → 3.7.1

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

Help available at : https://docs.python.org/2.7/ and https://docs.python.org/3/

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
```

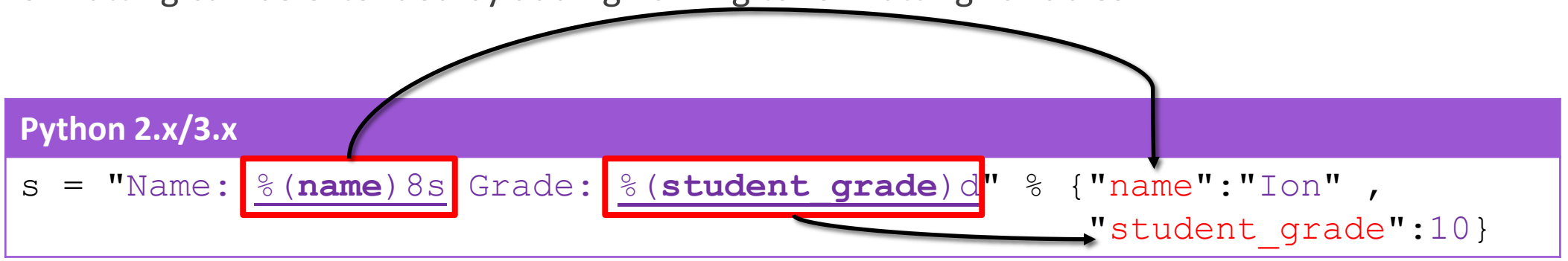
Two special keywords **str** and **repr** can be use to convert variables from any type to string.

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 EF||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

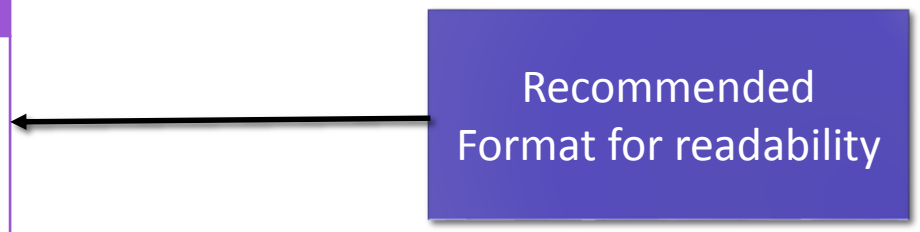
Statements

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.