Introduction to programming.

Lecturer : X. Parent (xavier.parent@uni.lu) ; G. Casini (giovanni.casini@uni.lu)

Teaching material: courtesy of Clément Guérin (previous lecturer) - slides freely adapted

Admin

When: Mondays 15h45- 17h15.

"Office hours" : Thursday 14:00-16:00.

This course will contain frontal sessions (13 lectures),

You will learn the basics of programmation. The chosen programming language is Python. It is a very good way to begin with programmation.

This is a learn-as-you-do course. Bring your laptop in class. We will do exercises!

There will be weekly homework assignments and a project. Projects presentation will be held during the last lecture.

The content of the lectures will be available on the moodle.

Evaluation

Grading

- 50% will come from weekly homeworks.
- 50% will come from a final project (starting in mid-November). Project presentations will be held on the last lecture

Rules for homeworks

- The homework assignments will be uploaded to the moodle the same day as the lecture.
- Deadline for submission: the Friday of the same week, 14:00. You should upload them on the moodle as file with an extension ".py" (executable file) or ".ipynb (notebook). In order to help us dealing with them, we strongly recommend you name them as HW{number of the Homework}_surname_name.{the extension you chose}.
- Mark range: 0 (min) - 20 (max)
- Homework not submitted in time (after the deadline) gets a 0.

Rules for projects

- They will be communicated in due course
- Mark range: 0 (min) - 20 (max)

Main evaluation criterions.

Please note that it is not exhaustive.

- Are the algorithms doing what they are supposed to do?
- Ergonomy of the program (use of **input**, relevant use of **print**...).
- Display of the source code. Am I able to understand the program from the source code? (Use of comments, good variable names, indentation,...).
- Are the algorithms efficient? If not, are the non-efficient algorithms highlighted as such with comments? Would it be possible to write another algorithm instead?
- In case the program does not work, we will carefully read any attempt. Explain and test your code, give your insight about what may be the problem and give partial results.
- (Projects) Insight and explanations in the pdf file.
- (Projects) Number of "black boxes" you used.
- (Projects) Writing skills (for the pdf file).
- (Projects) Specific time tests for crucial functions in the program or complexity evaluation.
- ...

Why learn programming?

The Industrial Mathematics track offers students the possibility to develop a skill set in mathematical techniques that are strongly needed in industry and were defined in cooperation with the Luxembourg Business Federation FEDIL.

If you go to industry, basic knowkedge of programming can be a plus. You'll have to interact with people with a very diverse background, including computer science people. In industry, AI is a hype.

What is Python?

It is a programming langage. It is also an interpreter. For compatibility reasons, I strongly recommend that we all work with the newest version of Python (3.6). You should also download and use the Anaconda navigator which provides a console, an editor and a notebook. That is all we are going to require for the lectures. Projects may require some specific tools.

Why Python?

(+ side) : Python langage is nice for beginners since it is high-level. In particular, the way to define functions, objects is meant to be easy for users.

The grammar of Python forces you to use one line for each command and to indent your code. In any langage, these are good habits. Indeed, you do want people to be able to read your source code without effort.

It is completely open access.

There are a lot of people using Python since it is easy to learn and easy to read. There are therefore many application modules (numpy, scipy, biopython,...) and much documentation too.

It is really easy to do collaborative projects with this langage.

(- side) : It is commonly assumed that, although it is quite efficient, Python langage is slightly slower than low-level langage such as C and C++.

```
from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = "all"
```

Useful websites.

==========

Downloads.

- Python : https://www.python.org (https://www.python.org)
- Anaconda : https://www.anaconda.com/download/ (https://www.anaconda.com/download/)
- Editors :
  - Emacs : https://www.gnu.org/software/emacs (https://www.gnu.org/software/emacs)
  - Atom : https://atom.io/ (https://atom.io/)
  - Gedit : https://wiki.gnome.org/Apps/Gedit (https://wiki.gnome.org/Apps/Gedit)

Documentation.

- (English) Official documentation in Python https://docs.python.org/3/ (https://docs.python.org/3/)
- (français) Lectures of M. Szopos, M. Mehrenberger, L. Navoret, P. Navaro for first year of graduate studies in Mathematics at the University of Strasbourg http://www-irma.u-strasbg.fr/~mehrenbe/slides_cours_python.pdf (http://www-irma.u-strasbg.fr/~mehrenbe/slides_cours_python.pdf)

How to use Python?

First Method. Interactive. (In Anaconda : qtconsole).

Like a calculator, you write a command, the console interprets it and send back the result when there is one to be sent. You may affect variables and define functions. As long as you don't close the window or does not shut down the kernel, the console keeps in its memory variables and functions.

(+ side) : Nice if you want to test a command or a function.

(- side) : You can't save, you can't go back.

Second Method. Execute a file. (In Anaconda : spyder).

You first write lines of command in a text file with a ".py" extension. This file may contain few lines up to thousands of them. Then you execute the file with the Python interpretor.

To create/write a file, you should use a text editor. Feel free to use the text editor you want. Spyder (in the Anaconda distribution) seems to work just fine. Here are some suggestions of text editors you can use...

With Windows : IDLE/spyder.

With Mac OS : IDLE/Atom/spyder.

With Linux : gedit/spyder/emacs.

Any software that create a text without hidden caracters is fine (Word would create some non-visible caracters). What is nice with the aforementioned editors is that they know the Python syntax and help you to not make mistakes in your writing.

Using spyder, you will be able to execute your code right away and use it with a console. You can also use your terminal and type in >>> python3 filename.py .

(+ side) : Your work can be saved. You can divide a big task into multiple files. Spyder allows you to use functions defined in your Python program in a console.

(- side) : Slightly less intuitive than the first method. It takes some time to get used to the loop write->execute->!bug!->debug->write->...

Third method. Notebook. (Dans Anaconda : Jupyter).

That's how I wrote this lecture. It allows you to go back and forth between texts, formulas (encoded in LaTeX) and lines of executable Python code.

(+ side) : You can expose your code with a text (lecture/project/work...). You can also convert the notebook as a pdf either by using LaTeX or convert it to html and then pdf. I would personally advise against using LaTeX because it does not look so nice. The ipynb to html to pdf method is much more faithful to the original look.

(- side) : Not really fit for doing long programs because it is slow.

Comments

Comments are used to add notesnand explanations to a program. In a large program, comments may describe the purpose of the program and how it works. They are solely intended for the people who are trying to understand the source code of the program. They are not programming statements so they are ignored by the Python Interpreter while executing the program.

In order to comment your source code, one uses the **hashtag** "#". It neutralizes everything after it on the same line.

In [ ]:
```
# This is a comment on a separate line
```

To make a longer comment, there is no other option that starting a new line, writing an hastag again and keeping on commenting

In [ ]:
```
a=1 # Not an important line.
a=1 # Not an import line, but I write
    # a lot of comments anyway. I don't
    # even know why I do that but I do
    # it anyway. Well you got the idea.
a=1 # New comment.
```

Basic math with Python

You may use Python as a calculator.

In [ ]:
```
2+2 #Addition
```

In [ ]:
```
2-19 #Substraction
```

In [ ]:
```
2*27 #Multiplication
```

In [ ]:
```
165/10 #Floating division.
```

In [ ]:
```
165//10# Quotient of the Euclidean division.
```

In [ ]:
```
165%10 # Remainder of the Euclidean division.
```

In [ ]:
```
3**2 # Power
```

In [ ]:
```
1==2# Equality test
```

In [ ]:
```
1!=2 # Non-equality test
```

In [ ]:
```
1<2 # less than
1>2# greater than
1<=2 # less or equal
1>=2 # greater or equal
```

Logical operators

In [ ]:
```
not(True) # the logical "NOT" operator
True and False # the logical "AND" operator
True or False # the logical "OR" operator
```

The "or" operator (like many built-in functions in Python) is optimized. Indeed, if you are evaluating P_1 or P_2 and P_1 happens to be true then P_1 or P_2 is evaluated as true before even evaluating P_2.

In [ ]:
```
inconnue==True #This statement makes no sense.
```

In [ ]:
```
1==1 or inconnue==True #The whole proposition is true
                       #even though "inconnue==True"
                       #is meaningless.
                       #Try the other way around.
```

Variables and types

Variables are containers for storing data values. Unlike other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it. The operator to do this is the symbol =.

In [ ]:
```
integer = 5 #An integer
real = 0.3 #A floating number
boolean= True # A boolean (True or False)
stringofcharacters = 'stringofcharacters' # A string of characters
between " " or ' '.
compl=1j
```

Each value has a type whence any affected variable has a type. This type may change throughout the program (dynamical typing).

In [ ]:
```
print(type(integer))
print(type(real))
print(type(boolean))
print(type(stringofcharacters))
print(type(compl))
```

Standard variable types

Python has five standard data types: Numbers ; String ; List ; Tuple ; Dictionary. Below we briefly explain the first two (more on this during lecture 2)

Numbers

Python supports two types of numbers: integers and floating point numbers. (It also supports complex numbers, which will not be explained in this lecture). To define an integer or a floating number, use the following syntax:

In [ ]:
```
int = 7
print(int)
```

In [ ]:
```
float = 0.5
print(float)
```

Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

In [ ]:
```
str ="Hello"  # To define a string you use this syntax
print(str)     # Prints complete string
```

In [ ]:
```
str ="7"
type(str)
```

In [ ]:
```
str=7
type(str)
```

The asterisk (*) is the repetition operator, and the plus (+) sign is the string concatenation operator

In [ ]:
```
str ="Hello"
print (str * 2)      # Prints string twice
print (str * 3)
print (str+"TEST") # Prints concatenated string
```

Rules for creating variable names

In Python, we have the following rules to create a valid variable name. Only letters (a-z, A-Z), underscore (*) and numbers (0-9) are allowed to create variable names, nothing else. Variable name must begin with an underscore (*) or a letter. You can't use reserved keywords to create variables names (see below). A variable name can be of any length.

Constants

Constants are variables whose values will not change during the lifetime of the program. Unlike languages like C or Java; Python doesn't have a special syntax to create constants. We create constants just like ordinary variables. However, to separate them from an ordinary variable, we use uppercase letters.

In [ ]:
```
MY_CONST = 100 # a constant
```

Note that MY_CONST is just a variable which refers to a value of type int. It has no other special properties. You are even allowed to change the value of MY_CONST constant by assigning a new value to it as follows:

In [ ]:
```
MY_CONST = "new value"
```

Value assignment

- = is not a symmetric operator: $v_1 = v_2$ does not mean the same thing as $v_2 = v_1$. This is because = assigns a value.
- Always read the formula from-right-to-left

In [ ]:
```
v1=2
v2=3
v1=v2 # v1  takes the value of v2
print("v1=",v1)
print("v2=",v2)
v1=2
v2=3
v2=v1 # v2 takes the value of v1
print("v1=",v1)
print("v2=",v2)
```

In [ ]:
```
v1=1
v1+=3 # Incrementing v_1.  It is equivalent to v_1=v_1+3
      # One should rather use  v1+=3 instead of the other
      # because it is more efficient (not only quicker to write).
print(v1)
v1*=2 #Also works with *=
print(v1)
```

In [ ]:
```
v1,v2=2,3 # Multiple affectations, the order matters!
print (v1,v2)
```

In [ ]:
```
v1=150
v2=3
v1,v2=v2,v1  # Python allows you to easily switch two variables.
print("v1=",v1)
print("v2=",v2)
```

In [ ]:
```
#To be compared with
v1=150
v2=3
v1=v2
v2=v1
print("v1=",v1)# Now v1 and v2 are equal.
print("v2=",v2)
```

I want to stress out that the type of variables you use really matters. You should always know the type of your variables at any step of your program. In order to do so, you have built-in functions that allows you to convert the type of your variables. Indeed, if you want to change the type of v to **ftype**, then you should call $v = \mathbf{ftype}(v)$. The effect strongly depends on the beginning type of the variable. You should test it on some types we already saw. For instance, if you apply **int** to a floating number then you get its integer part.

In [ ]:
```
print(integer)
print(float(integer))
print(real)
print(int(real))
```

Some variable names are protected. For instance, if you try to do the following, you will have an error message.

In [ ]:
```
True=3
```

Not all names are protected. You should be careful about this. Here are some rules that I usually follow to name my variables :

- make sure that the name has not been chosen before or that you don't need the former variable anymore (the text editor can help you with this issue).
- only use one letter names for variables that you don't need to keep track of.
- for important variables, use specific names.
- don't use too long names since they tend to make the code less readable.

Finally, here is a list of protected names in Python to be compared with the list of already built-in functions. I strongly recommend to avoid naming a variable after any of those names.

In [ ]:
```
import keyword
for x in keyword.kwlist:
    print(x,end=" - ")#This is the list of all protected names.
print()
print()
import builtins
for x in  dir(builtins): #Functions that are already built.
    print (x, end= " - ")
```

Functions

In Python, a function is called like this $\mathbf{function}(variable_1, \dots, variable_r)$. We have already seen some functions such has **type**, **int**,... Above, we have a list of already built Python functions. Later we will be able to build our own functions. For now, we focus on a few examples that are useful.

Input

With the input() function the program gets input from the user. The **input** function takes a string of characters as an argument. Then, the console write this string of characters along with an invit to enter something with the keyboard which is then the value of the input function. The type of the result is always a string of characters.

In [ ]:
```
n=input("Please enter a number : ")
```

In [ ]:
```
n=input("Please enter a number : ")
type(n)# Watch out! The type of n is "str" not "int".
```

In [1]:
```
n=input("Please enter a number : ")
n+'1'  # Here is what happens when we play with the wrong type.
```

```
Please enter a number : 12
```

Out[1]:
```
'121'
```

Here is how to change the type of the input from str (string) to integer (int). This is needed to do arithmetical operations on the input

In [3]:
```
n=int(input("Please enter a number : "))
print(n>=12)
```

```
Please enter a number : 13
True
```

The print function

In [ ]:
```
print(3) # It prints the integer 3
print('dies irae') # It prints dies irae
print(True) # It prints True
```

In [1]:
```
x=2
print(x) # It prints the value of a variable and not its name.
```

```
2
```

The **print** function can have as many arguments as you want. Whether the arguments are values or variable names, it will print their values, in the same order, separated by a space. When using Python executing a .py file and note in interactive nor in notebook modes, it is necessary to use the **print** function to make something actually appear on the screen.

As a default parameter, different lines of **print**$(\cdot)$ commands are printed on different lines.

In [ ]:
```
print(3,'dies irae',True) # you may print different arguments
                          # on the same line.
```

If you want to change the end of the print command which is by default equal to \n (i.e. line break), you should add another argument : end="whichever string you want".

In [ ]:
```
print(3,end="*")          # Doesn't break the line but put a * inst
ead.
print('dies irae',end="*")# Doesn't break the line but put a * inst
ead.
print(True)               # Breaks the line.
print(3,end="   ")          # Doesn't break the line but put a spac
e instead.
print('dies irae',end="   ") # Doesn't break the line but put a spac
e instead.
print(True)                 # Breaks the line.
```

The help function

In [ ]:
```
help(print)
help(input)
help(help)
```

The **help** function takes any (built-in) function as an argument and send back its code.

Some rules to write your code in Python.

Any computer langage requires an absolute rigor when you write it. In general, when you do some mistake, you directly get an error message, in some few cases you will just end up with a computation case but it is rarely harmless. We recall that in Python the grammar is also supposed to help you to write a readable source code.

In Python langage, the execution of a .py file will go through each line of your codes, and will execute them one after the other.

Rule 1. To end a command line you go to the next line i.e. use a line break (whereas some other langages would end a command by ";", ":" or "::"). **Line break has a meaning in Python**. It is therefore impossible to write two commands on a same line.

In [ ]:
```
a=3  b=2
```

In [ ]:
```
a=3
b=2
```

In the following example, we write a perfectly valid line of command. However it is a very long one and it makes it painful to read on the source code.

In [ ]:
```
print('Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut e
nim ad minim veniam, quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat. Duis aute irure dolor in repreh
enderit in voluptate velit esse cillum dolore eu fugiat nulla paria
tur. Excepteur sint occaecat cupidatat non proident, sunt in culpa
qui officia deserunt mollit anim id est laborum.')
```

The immediat solution is to use line breaks in the source code. But since line breaks have a meaning, the interpretor understands it as the end of the command which is incomplete (the string of characters has no end, no closing parenthesis,...) whence an error arises.

In [ ]:
```
print('Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
do eiusmod tempor incididunt
      ut labore et dolore magna aliqua. Ut enim ad minim veniam, qu
is nostrud exercitation ullamco
      laboris nisi ut aliquip ex ea commodo consequat. Duis aute ir
ure dolor in reprehenderit
      in voluptate velit esse cillum dolore eu fugiat nulla pariatu
r. Excepteur sint occaecat
      cupidatat non proident, sunt in culpa qui officia deserunt mo
llit anim id est laborum.')
```

Rule 2. The good way to deal with this problem is to neutralize the "line break" character by adding a backslash "\" before breaking the line. Of course, if you put something right between the "\" and the next line you will neutralize what you put in between instead of the line break! So be careful about that.

In [ ]:
```
print('Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
\
do eiusmod tempor incididunt ut labore et dolore\
magna aliqua. Ut enim ad minim veniam,\
 quis nostrud exercitation ullamco \
laboris nisi ut aliquip ex ea commodo \
consequat. Duis aute irure dolor in reprehenderit\
in voluptate velit esse cillum dolore eu fugiat nulla pariatur. \
Excepteur sint occaecat cupidatat non proident,\
sunt in culpa qui officia deserunt mollit anim id est laborum.')
```

Rule 3. **Python is an indented langage**. Basically it means that all lines of command beginning at the same place will have the same logical status. Another way to put this
spaces at the beginning of a line have a meaning (except when one uses the hashtag).

In [ ]:
```
2+3 # No problem
```

In [ ]:
```
 3+2 # No problem either
```

In [ ]:
```
2+3
 3+2 # Here, the first and second line should
     # have the same indentation because there
     # is no change in the dynamical flow such as
     # loops, if statements, definitions...
```

Rule 5. To use conditional statements, one should use a colon ":". The different words ponctuating the if statement are **if**, **elif** and **else**. You put ":" after each occurrence of **if**, **elif** and **else**. The else part is optional. **elif** is short for "else if". Be careful with the indentation. The structure is like this:

```
if :
   [do something]
   ....
   ....
elif [another statement is true]:
   [do something else]
   ....
   ....
else:
   [do another thing]
   ....
   ....
```

In [ ]:
```
if 0==1 # Don't forget the colon.
    print('Problem')
else:
    print('No Problem')
```

In [ ]:
```
if 0==1:
print('Problem')# Don't forget the indentation
                # It is usually completely automatic
                # with smart text editors.
else:
    print('No Problem')
```

In [ ]:
```
if 0==1:
    print('Problem')
    else: #"if", "elif" and "else" should have the same indentation
.
    print('No Problem')
```

In [ ]:
```
if 0==1:
    print('Problem')
else:
    print('No Problem')
```

In [ ]:
```
#A not really smart example with many different conditions.
n=int(input("Enter a number  :  ")) # We enter a number n
if n%7==0:   # Same indentation level for "if", "elif" and "else".
    print(n, "is congruent to 0 modulo 7")
elif n%7==1:
    print(n, "is congruent to 1 modulo 7")
elif n%7==2:
    print(n, "is congruent to 2 modulo 7")
elif n%7==3:
    print(n, "is congruent to 3 modulo 7")
elif n%7==4:
    print(n, "is congruent to 4 modulo 7")
elif n%7==5:
    print(n, "is congruent to 5 modulo 7")
else:
    print(n, "is congruent to 6 modulo 7")
# Smart way to do this.
n=int(input("Enter a number : "))
print(n," is congruent to ",n%2713,"modulo 2713")
```

In [ ]:
```
#An example with different if statements one in the other.
n=int(input("Enter a number between 8 and 49 to test its primality
: "))
isprime=True
if n<7:
    print("Impossible! the number is too small.")
elif n>49:
    print("Impossible! the number is too big.")
else:
    if 2*(n//2)==n:
        print("Not prime, ", n," is divisible by 2.") # There are 3
levels of indentation.
       isprime=False
    elif 3*(n//3)==n:
        print("Not prime, ", n," is divisible by 3.")
        isprime=False
    elif 5*(n//5)==n:
        print("Not prime, ", n," is divisible by 5.")
        isprime=False
    elif 7*(n//7)==n:
        print("Not prime, ",n," is divisible by 7.")
        isprime=False
    else:
        print(n," is prime.")
```

While loops

A **while** loop statement repeatedly executes a command as long as a given condition is true.

The syntax of a while loop is

```
while expression:
   statement(s)
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop. In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

In [ ]:
```
count = 0
while (count < 9):# We state the condition under which we stay in t
he loop.
   print('The count is:', count) #We do something
   count = count + 1 #We do something else

print("Good bye!") #We do this when we're outside the loop
```

In [ ]:
```
N=int(input("I will count up to : "))
p=0
while p<N: # We state the condition under which we stay in the loop
.
    p+=1# We do something
    print(p,end=" ") # We do some other thing.
```

Please note that, like for the conditional statement, you need to end the **while** line of command with a colon and that you need to indent the block of commands that will be repeated in the loop.

In general, when you define a **while** loop, you need to make sure that the loop will stop at some point. Indeed, if you forget this you might never go out of your while loop in which case, if you want to be in charge again, you need to restart the kernel.