Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Introduction;
Functions andData Types (in Python)

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2020/2021

Introduction

- This course introduces the basic concepts of
  - Programming,
  - Algorithms; and
  - Databases
- The language adopted is **Python**
  - To introduce basic programming concepts and algorithm techniques, with an interface to sqlite (databases)
- The database tool used in sqlite
  - To introduce the topic of databases and the language SQL

- All information about the course (synopsis, lectures, exercises, rules, lecturers, etc.) is maintained in the web page
  - **http://mc.ssdi.di.fct.unl.pt**

Programs and Algorithms

- In very abstract terms, a computer program can be regarded as a set of instructions that applied to input data yields some result as output.

- An algorithm is a sequence of instructions, written in a programming language, understood by some executor.
- Programs and data are stored in computers, that are able to execute the stored programs on the data (input or stored) to produce other data, eventually sent to the output.

Programming Languages

- A program may be specified at different levels of abstraction. The more knowledgeable the executor is, the more high level the program may be specified.
- At the lowest level, such information is coded in binary digits (bits) and stored somewhere in the memory of the computer. Processing it requires the execution of machine instructions (almost directly encoded in assembly) that command the computer to move and transform the bit-encoded data.
- For example, one may instruct to add the data stored in memory positions 1001 and 1002, and store the result in position 1003, by means of the assembly code below

```
LDA 1001
LDB 1002
ADD A,B
STA 1003
```

- Of course, specifying a program at this low level requires the user to keep track of all the details regarding the encoding of data, the positions in memory it is stored in, etc., which is very “unpractical”.

Programming Languages

- High level languages allow the user to specify the programs in a more human readable form, so easing the task of programming.
- For the previous example, where we wanted to sum two numbers, high level languages allow this instruction to be specified as (or similarly to)

```
C = A + B
```

- Of course, in this case, the executor of the program must take care of all the details mentioned before, namely that the memory positions where the data corresponding to the numbers A, B and C is stored.

Programming Languages

- In fact there are several types of high-level programming languages with different features namely:
- **Imperative Languages** : Fortran, Pascal, C, *Octave/MATLAB*
  - With an explicit control of execution
- **Object Oriented Languages**: Smalltalk, C++, Java, **Python**
  - Allowing abstraction of data objects, with arbitrarily complex properties.
- **Functional Languages**: LISP, Scheme
  - Programs are specified as functions, rather than “instructions”
- **Logic Languages**: Prolog, ASP
  - Programs are specified as predicates of 1st order logic
- **Database Languages**: SQL
  - Mixing features of the above, to query databases.

IDE - Interactive Development Environment

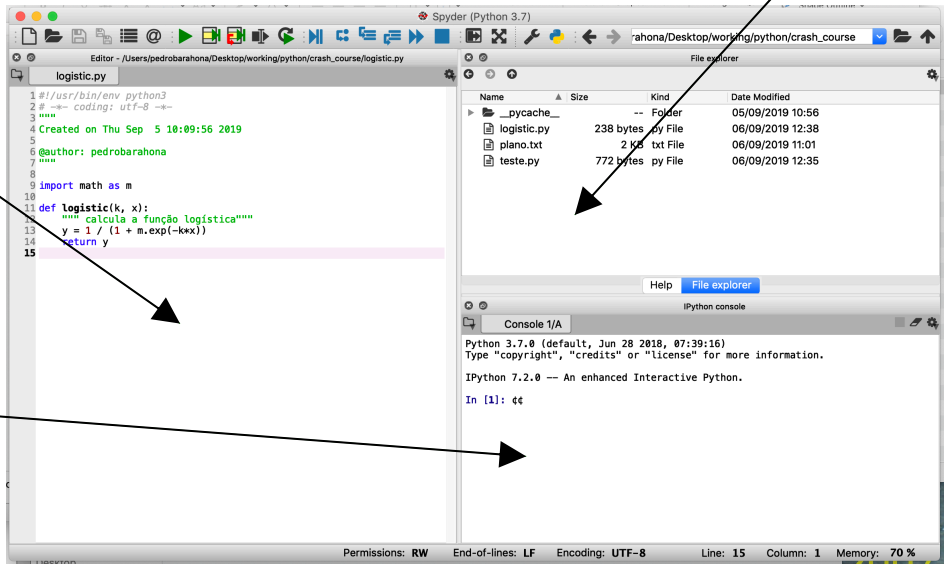
- In practice, programming at high level requires several “programs” namely:
  - An **editor** – to write down the programs and store them in text files.
    - Currently, these editors are often included in IDEs (Integrated Development Environment) that help organizing and inspecting different components of the programs.
  - An **executor** – to interpret and execute the instructions
- There are two main types of executors to deal with programs:
  - **Compilers**: Whole programs (or rather components) are translated into machine language, producing an executable file (e.g. and exe file) that can be executed from the operating system.
  - **Interpreters**: Programs are executed instruction by instruction, from the IDE (that can be as simple as a console) and no executable file is produced.
- In this course we will adopt language **Python**, and use the **Anaconda/Spyder** IDE (free download – instructions in the course web page).

Spyder IDE

- Once installed the Anaconda System, its Spyder IDE shows 3 windows: Editor, File Explorer and Console.

The **File Explorer** allows the selection of the **Working Directory**. To simplify, this should be the directory where all the programs and data files are stored.

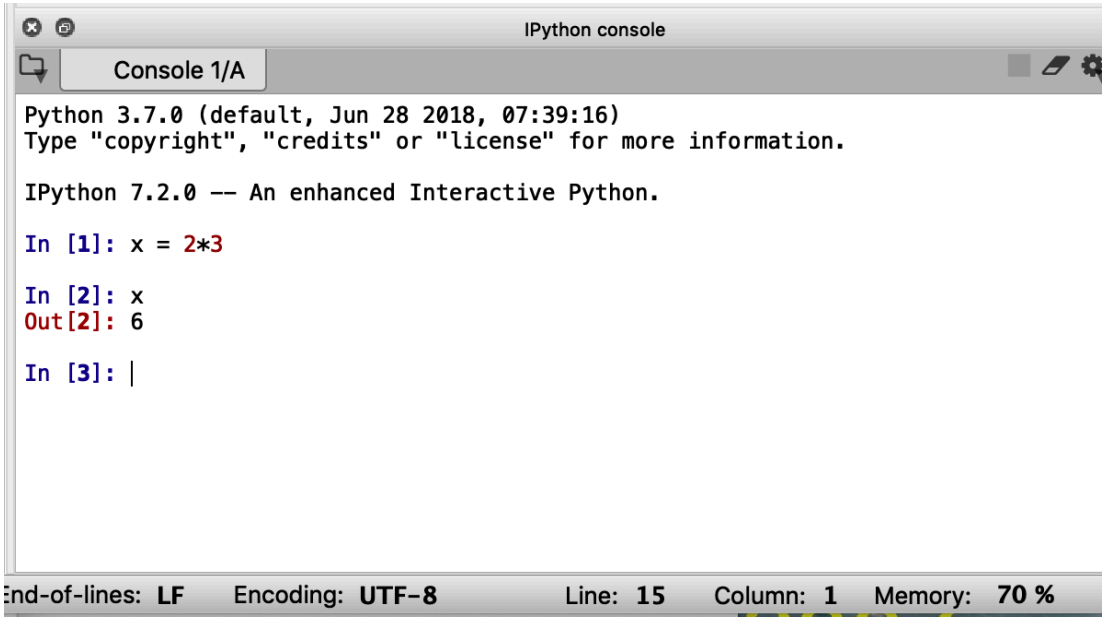
The **Editor** allows the edition (open, write, update and close) of the programs and data files to be developed.



The **Console** allows the direct interaction with the user

Spyder IDE

- The Console may be used as a simple calculator, using Python instructions.



- We should now introduce the simples instructions (in Python).

Basic Instructions

- In the imperative programming paradigm, the programmer specifies explicitly the control of execution, i.e. the sequencing of the basic operations.
- Informally, we may consider the following basic instruction in a high-level language like Python.
  - **Assignment**: `V = Expression`
    - Variable **V** takes the value of the **Expression.**
  - **Input**: `Input V`
    - Variable takes a value read from some external device (e.g. from a file)
  - **Output**: `Output V`
    - Variable V is written to some output device (e.g. to a file, or a printer).
- Although in most cases values are simple numbers, assignment operations can be used with more complex data structures and with data of different types.
- The exact syntax and the data that can be involved in these operations depend on the programming language adopted. With no loss of generality, we will adopt the Python language in this course.

Data Types

- In assignment operations several types of data can be used. The simplest data types, as used in Python, are
  - Numeric (Integer and Float)
  - Boolean (True / False)
  - Characters
- Notice that these types are mostly relevant from a programming language perspective. At machine level, they are all encoded into sequence of bits (e.g. 16, 32 or 64 bits) stored in memory, depending on the architecture of the computer being used.
- The executor of the language should be able, not only to keep track of the memory positions where the data is stored, but also to interpret the corresponding bits.
  - For example, byte **00101000**, can be interpreted as the character ASCII character "**(**", or the integer **40**.
- In many languages, mostly to be compiled (as Java, but not Python) the programmer must explicitly declare the type of the variables to be used, so that not only they may be efficiently encoded but also to aid a compiler to debug a program.

Assignment Instruction

- Assignments instruction such as

```
V = Expression
```

  assign the value of the right-hand expression to the left-hand variable. Hence the left hand **must always be a variable**.

  - Notice that the = sign does not represent equality, but rather assignment!

- **Syntactically**, a variable is denoted by a sequence of letters (including _) and digits, started by a letter (note that upper and lower case letters are considered different)
- As to the expression, they are composed of operations on both variables, constants or functions (later) of the appropriate type.
- For example, **numeric variables** can be assigned with expressions including the usual arithmetic operators (sum, subtraction, multiplication, division and exponentiation) possibly using parenthesis to take into account the usual precedence of operators.

  - Notice that division is always real division, even when applied to integer values.

Numeric Expressions

- The following examples, with the usual arithmetic operators, sum (+), subtraction (-), multiplication (*), division (/) and power (**), can be tested in the Spyder console.

```
In : 3 + 7;
Out: 10
In : a = 3 * (4 + 6**2) - 6
In : a
Out: 114
In : 5**0
Out: 1
In : 5.0**0
Out: 1.0
In : 12 / 3
Out: 4
In : 12 / 0
ZeroDivisionError: division by zero
```

- Note that assigning a value to a variable does not display the (new) variable value. To see it one must type the variable name.

Numeric Expressions

- Other operators and some useful arithmetic **functions** are available in Python
  - **`abs/1, round/1`**
    - the absolute value and the rounding of the argument
  - **`//2, % 2`**
    - the quotient of the integer division
    - the remainder of the integer division

```
In : abs(-4.6);
Out: 4.6
In : round(5.9)
Out: 6.0
In : 7 / 2
Out : 3.5
In : 7 // 2
Out: 3
In : 7 % 2
Out: 1
In : 7.4 % 2
Out: 1.4000000000000004
```

Numeric Expressions

- More “complex” mathematical functions are provided in library **math**.
- To be used, a library must be imported, with instruction **import**, possibly assigning it a short name to use as a **prefix**.
- Once imported, functions defined in the library may be used, using the chosen prefix.
- The functions defined in the library can be consulted with instruction dir() (although using the references available in the web are usually more informative).

```
In : import math as m
In : m.cos(0)
Out: 1.0
In : m.cos(m.pi/6)
Out: 0.4999999999999994
In : dir(m)
Out:
['__doc__’,
 ...
 'acos’,
 ...
'trunc']
```

Numeric Functions – The Math Library

- The most common mathematical functions are available in the library
  - **`ceil/1, floor/1, trunc/1`**
    - Conversion of reals into integers
  - **`factorial/1`**
    - the factorial of an integer
  - **`sqrt/1`**
    - the square root of an number
  - **`log/1, log10/1, log2/1`**
    - the logarithms (natural, base 10 and base 2)
  - **`exp/1`**
    - the exponential functions (base e and base 10)
  - **`sin/1, cos/1, tan/1, asin/1, acos/1, atan/1`**
    - the trigonometric functions (arguments in radians)
  - **`pi`** and **`e`**
    - The constants
  - **`inf`**
    - The “infinite”

Numeric Expressions

- Examples:

```
In : import math as m
In : m.ceil(3.9)
Out: 3.0
In : factorial(5)
Out: 120
In : m.log10(100)
Out: 2.0
In : m.log2(128)
Out: 7.0
In : m.exp(m.log(56))
Out: 56.00000000000002
In : m.cos(m.pi)
Out: -1.0
In : m.atan(m.inf)
Out: 1.5707963267948966
In : m.degrees(m.atan(m.inf))
Out: 90.0
```

Boolean Expressions

- **Simple Boolean expressions** are obtained with the relational operators
  - == , != equality and disequality
  - >=, =< inequality
  - > , < strict inequality
  - That evaluate to one Boolean value, **True** or **False**.

- More complex Boolean Expressions may be othined with the Boolean operators
  - and conjunction,
  - or disjunction; and
  - not negation

- **Note**: Boolean expressions are mostly used to express conditions in both conditional (if) and iterating (while) instructions.

Boolean Expressions

- Examples:

```
In : a = 3 > 5
In : a
Out: False
In : m.pi
Out: 3.141592653589793
In : m.pi == 3.1415926535897
Out: False                  # beware of precision
In : abs(m.pi - 3.1415926535897) < 0.00001
Out: True                   # deal with precision
In : a = 6
In : a < 3 or a > 20
Out: False
In : not(a < 3 or a > 20)
Out: True
In : not (a < 3) and not (a > 20)
Out: True
```

User Defined Functions

- In addition to functions defined in the standard libraries, users may define their own functions. These must be defined in a text file, which can be edited in the Spyder Editor. Functions have the following syntax:

```
def FunctionName(parameters):
        """ Function documentation
        """
        function block
        return [None | result(s)]
```

- The signature specifies the name of the function and its parameters.
- The **parameters** are input for the function and are separated by commas (if there are none, the function is a constant).
- **The function** documentation describes the purpose of the function.
- The **function block** is a set of instructions, adequately **indented** and **commented**, that manipulate the input parameters and internal variables and possibly yield some result(s).

User Defined Functions

```
def FunctionName(parameters):
    """ Function documentation
    """
    Instruction block
    return [None | result(s)]
```

- The value of the function is specified with the **return** instruction.
- When the return instruction is executed, the function is finished, and control resumes to the program that called it.
- A function might return no results and be used only to produce some side effect.
  - For example, function print() does not return any value, but only shows the results in the console.
- **Note:** Although optional, the documentation is very useful to elucidate the purpose of the function and is available with the command
  - help(functionName)

User Defined Functions

- Example:

```
def inside(x, a, b):
    """
    this function indicates whether x lies inside the
    interval [a,b].
    """
    between = (x >= a) and (x <= b)  # performs the check
    return between
```

- The function computes a Boolean expression, and assigns it to a local variable, **between**, that is then returned.
- The local variable between is not accessible from outside the function, and may be omitted in this case, so as the expression is returned directly as

```
    return (x >= a) and (x <= b)  # performs the check
```

Import / Load Defined Functions

- Once defined the functions may be used after performing one or two commands on the file where the function is defined:
  1. Importing the file (i.e. using it as a user defined library)
  2. loading the file (i.e. executing the load instruction)

Import

- The file may be imported, similarly to what was done with the math library.
- In this case, the functions in the file must be prefixed by the file name (or its alias).

Load

- The file may be loaded (executed), in which case, the definition of the functions are considered in the calls, *as when they are loaded*, with no need to use a prefix..
- Notice that in this case, any change to the file, e.g. to correct a bug, requires the file to be loaded again!
- Loading a file may be done in Spyder by clicking on the load button: 

User Defined Functions

- Example: Function inside is defined in a file with name **my_functions.py**

```
In : inside(5, 1, 8)
NameError: name 'inside' is not defined
In : import my_functions as mf
In : inside(5, 1, 8)
Out:
NameError: name 'inside' is not defined
In : mf.inside(5, 1, 8)
Out: True
In : x = mf.inside(8, 1, 8)
In : x
Out: False
In : runfile(...)
In : inside(5, 1, 8)
In : True
```

Scripts

- A sequence of instructions that is executed often (e.g. for debugging) may be stored in a script, so that the use does not have to type it time and again.
- A script is just a file, with extension .py, where the instructions are stored.
- When loading the script file, the instructions are executed, as if they were typed in the terminal.

My_script.py

```
a = 1
b = 3
```

```
In : runfile(...).
In : a
Out: 1
In : c = a + b
In : c
Out: 4
```

Python Lists - Arrays

- In most applications information is naturally structured in arrays, and programming languages have support for these data structures.
- In general, arrays may be uni-dimensional (vectors), bi-dimensional (matrices) or of higher dimensionality.
- Although Python offers a specialised library (NumPy) to optimally deal with all these arrays, and for didactic purposes, we will use Python lists to implement these data structures (of course, students are invited to see into it, either during or after the course).
- Lists are created by specifying a sequence of items in square brackets.
- Note that the items in the list can be of any type. To create a numeric vector by means of a list, the items must all be numeric.
- A two-dimensional array (e.g. a matrix) may be implemented as a list of lists.
- Of course, to represent a matrix all the lists must have the same number of elements.

Vectors and Matrices as Python Lists

- Examples:

```
In : # an arbitrary list
In : A = [ 4, 7, 5, "any"]
In : A
Out: [4, 7, 5, 'any']
In : # an empty list
In : V = [ ]
In : V
Out: []
In : # a vector with size 3, implemented as a list
In : V = [1,2,3]
In : V
Out: [1,2,3]
In : # a 2 x 3 matrix implemented as a list of lists
In : M =[[1,2,3],[4,5,6]]
In : M
Out: [[1,2,3],[4,5,6]]
```

Accessing Elements of Python Lists

- Once created, individual elements of a list can be accessed by their index (in square brackets).
- In a list with n elements, indices range from 0 to n-1, where the first element as index 0 and the last index n-1.
- Alternatively, the indices can be assigned in reverse order. In this case, index -1 corresponds to the last element of the list, and index –n corresponds to the first element of the list.
- When an array is composed of lists of lists, two indices must be provided to access the element:
  - The first index indicates the list to which the element belongs;
  - The second index indicates the position of th
- This method applies to structures with any number of dimensions. To access any element of a 3-dimensional array (a list of lists of lists) 3 indices are required.

Accessing Elements of Python Lists

- Examples:

```
In : V = [1,2,3]
In : V[1]
Out: 2
In : V[0]
Out: 1
In : V[-1]
Out: 3
In : V[-3]
Out: 1
In : M =[[1,2,3],[4,5,6]]
In : M[1][2]
Out: 6
In : M[-2][-1]
Out: 3
In : M[0][2]
Out: 3
```

Slices in Python Lists

- More than one element of a list can be selected. A **slice** of a list includes
  - The first element to be selected, **inclusive**;
  - The last element to be selected, **exclusive**;
  - The gap between consecutive elements.

- Notice, that
  - Gaps may be omitted (by default, the gap is 1).
  - Gaps may be negative (the elements are seleted “backwards”)
  - The slices may be such that no element is seleted (for example, if the gap is positive and the first index is greater or equal to the second).

- Care must be taken to use valid indices. Accessing individual elements with an invalid index raises an “out of bounds error”, interrupting the computation.
  - When “wrong” indices are used in slices, the empty list is returned.

Slices in Python Lists

- Examples:

```
In : V = [1,2,3,4,5,6,7]
In : V[0:3]
Out: [1,2,3]
In : V[0:5:2]
Out: [1,3,5]
In : V[-1:-3]
Out: []
In : V[-1:-3:-1]
Out: [7,6]
In : V[6:4:-1]
Out: [7,6]
In : V[9:7]
out: []
In : V[7:9]
out: []
In : V[7]
out: IndexError: list index out of range
```