Functions; IF and FOR instructions

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2019/2020

Conditional Execution - IF

- The function **angle/2** that was discussed in the (and is shown below) executes a sequence of assignment instructions, some of them calling pre-defined functions, like **sqrt/1** and **acos/1**), as well as other user defined functions (e.g. **length/1** and **dot_product/2**).

```
def length(u):
   """Returns ... """
   return m.sqrt(u[0]**2 + u[1]**2 + u[2]**2)

def dot_product(u,v):
   """Returns ... """
   dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2]
   return dot

def angle(u,v):
    """Returns ..."""
    c = dot_product(u,v) / (length(u) * length(v))
    return m.acos(c)
```

- This is a very rare situation. In most programs/functions **the sequence of instructions depends on conditions of the data being used**.

Conditional Execution - IF

- For specifying this conditional execution, all languages include an instruction: **IF**. Syntax may vary for different languages so here we will use the Python syntax.
- In its simplest form this instruction conditions the execution of a THEN-BLOCK, where the CONDITION is any Boolean Expression.

```
if <CONDITION>:
    THEN-BLOCK
```

- Very often the instructions selects one of two sequence of instructions: either the THEN-BLOCK or the ELSE-BLOCK is executed

```
if <CONDITION>:
    THEN-BLOCK
else:
    ELSE-BLOCK
```

- IMPORTANT: Notice that the THEN- and ELSE- blocks must be **indented** wrt the if / else declaration. Moreover, the else keyword must be aligned with the if keyword

Conditional Execution - IF

- We may illustrate the first case with a function to compute the absolute value of a number (in fact this function is already pre-defined, as **abs/1**).

```
def absolute(x):
    """ returns the absolute value of x """
    a = x
    if x < 0:
      a = -a  # changes the sign of a
   return a
```

- An alternative specification of this function would use the else statement

```
def absolute(x):
   """ returns the absolute value of x """
   if x < 0:
      a = -x
   else:
      a = +x
   return a
```

Conditional Execution - IF

- A more complex example: Find the (real) roots of a 2nd degree equation

```
def equation_2(a, b, c):
    """ returns the solutions of equation
    ax^2 + bx + c = 0 (assuming a != 0)
     """
    d = b**2 - 4*a*c;
    if d < 0:                          # no solutions
       roots = []                      # roots is an empty vector
    else:
       if d == 0:                      # one single solution
          roots = [-b/(2*a)]
       else:                           # two distinct solutions
          roots = [-b + m.sqrt(d) / (2*a),
                   -b - m.sqrt(d) / (2*a)]
     return roots
```

- **Note 1**: Notice the indentation – otherwise the code is WRONG.
- **Note 2**: Notice the comments – they make the code more "understandable"

Conditional Execution - IF

- The previous example illustrates the “nesting” of if statements (if inside an if blocks).
- The code becomes more readable if one uses not a single ELSE-BLOCK but several ELIF-BLOCKS.

```
function equation_2(a, b, c):
    """ returns the solutions of equation
    ax^2 + bx + c = 0 (assuming a != 0)
     """
    d = b**2 - 4*a*c;
    if d < 0:                       # no solutions
       roots = []                   # roots is an empty vector
    elif d == 0:                    # one single solution
         roots = [-b/(2*a)]
     else:                          # two distinct solutions
         roots = [-b + m.sqrt(d) / (2*a),
                  [-b - m.sqrt(d) / (2*a)]
     return roots
```

Python – Lists

- Before addressing the FOR instruction for repeated execution of a block of instructions, we note that this instruction is often associated to Lists and other data strauctures, that we overview here.
- As seen before, Python provides the data structure **list**, to allow the organization of collection of any type of objects, not only of simple data types (e.g. Int or float) but also other more complex objects, such as lists.
- Instances (objects) of this type of data structure (class) are typically created with simple enumeration. For example,

```
In : L = [1,2,3,4]
In : M = [1, "a", [1,2,3]]
In : S = ["a", "b", "c"]
```

- The last case, a list of characters is usually created as as string,

```
In : S = "abcd"
```

Python – Lists

- Before using a list, it is convenient to initialise it, which can be done with the repetition instruction.

```
In : L = [0]* 5
In : L
Out: [0,0,0,0,0]
In : [None]*3
In : [None, None, None]
```

- They can also be initialised by comprehension (ranges come next)

```
In : L = [i*2 for i in range(3)]
In : L
Out: [0,2,4]
```

- Lists are ”mutable” objects, in that they can be appended with extra elements, extended with other lists, or have elements removed.
- Methods for list objects are available to perform these changes.

Python – Lists

- In general, existing methods available for an object may be consulted with the dir command.

```
In : L = [0]* 5
In : dir(L)
Out:
['__add__’,
   .....
 '__len__’,
   .....
 'append’,
 'copy’,
 'extend’,
   .....
 'remove',
 'reverse',
 'sort']
```

Python – Lists

- Some examples:

```
In : L = [1,2,3,4]
In : M = [6,8,7,8]
In : L.append(5)
In : L
Out: L = [1,2,3,4,5]
In : L.extend(M)
In : L
Out: L = [1,2,3,4,5,6,8,7,8]
In : L.remove(8)
In : L
Out: L = [1,2,3,4,5,6,7,8]
```

Python – Lists

- Lists are not sets, in that elements of the list have a position (index).
- Indices in a list of length n, range from 0 to n-1. Elements of a list can be accessed by means of their index, either positive (0 to n-1, from left to right) or negative (from -1 to –n) from right to left.
- The length of a list can be obtained with method **len**.

```
In : L = [1,2,3,4]
In : len(L)
Out: 4
In : L.__len__()
Out: 4
In : L[2]
Out: 3
In : x = L[-3]
In : x
Out: 2
```

Python – Lists

- Lists are **mutable** objects in that their state may change.
- Not only the lists can be extended and "shrinked" as seen before, but also their elements may change.

```
In : L = [1,2,3,4,5,6]
In : L[3] = 9
In : L
Out: [1,2,3,9,5,6]
```

Python – Tuples

- Tuples are similar to lists. They can be created by enumeration with brackets notation.
- However, tuples are **immutable** objects. Once created they can not be changed.

```
In : T = (1,2,3,4,5,6)
In : T[1]
Out: 2
Out: T[1] = 9
TypeError: 'tuple' object does not support item assignment
```

- Methods available to tuple objects can be obtained with the command **dir**.

Python – Sets

- Sets are also similar to lists, but
  - their elements are not accessible by indices.
  - they do not take repeated elements.

```
In : S = {1,2,3,1}
In : S
Out: {1,2,3}
Out: S[1]
TypeError: 'set' object does not support indexing
```

- Methods available to set objects can be obtained with the command dir.
- Sets are useful to implement dictionaries (later).

Python – Matrices

- Matrices (and higher order arrays) can be implemented as lists of lists.
- Their elements can be reached as before, but now there are to indices to consider
  - An index for the rows
  - An index to the columns

```
In : M = [[1,2,3,4],[4,5,6,7]]
In : len(M)     # number of rows
Out: 2
In : len(M[0]) # number of columns
Out: 4
In : M[1][2]
Out: 6
```

- Although all matrix operations can be implemented with nested lists, library **NumPy** is very useful for linear algebra operations on vectors and arrays (later).

Iterative Execution - FOR

- In many cases it is necessary to repeat a block of instructions. There are several variants to specify such repetition, and the simplest one is with a FOR statement.
- In Python syntax

```
for ITERATION-VAR in ITERATOR:
   FOR-BLOCK
```

- This instruction specifies that the FOR-BLOCK
  - is executed as many times as there are elements in the ITERATOR;
  - In each execution the ITERATION-VAR takes the value of the corresponding element of the ITERATOR;
  - Note: The ITERATION-VAR is usually used in the FOR-BLOCK, although this is not necessary

Iterators - Ranges

- In Python there are several types of iterators.
- **Lists / Tuples / Sets**: Common iterators are lists, tuples and sets. In this case, the iteration variable takes all the values of the list / tuple / set, one for each iteration.
- Fo example, the following snippet prints all the values of a list.

```
V = [1, 3, 5]
for i in V:
    print(i)
```

- **Ranges**: Another often used iterator is a **range**. It can be regarded as a generator of a list, by specifying the **first** value, the **limit** value (excluded), and the **step**.
- For example, the same behaviour obtained above would be produced with the code:

```
for i in range(1,6,2):
    print(i)
```

Iterators - Ranges

- The general specification

```
range(first, limit, step)
```

- generates consecutive elements starting at first (a number), continuing with all values obtained by adding the **step** (a number, different from zero) to the previous value as long as the limit is not reached, i.e. the last element must be **before** that limit.
- When the step is **1** it may be omitted.
- When the first value is **0**, it may also be omitted. The following ranges are equivalent

```
range(0,5,1)
range(0,5)
range(5)
```

- Ranges (as lists or sets) can be empty, when the first element is greater than the limit. This is the case of

```
range(4,4,2)
range(5,4)
```

Iterators

- Ranges can also generate decreasing values if the step is negative.

`range(first, limit, -step)`

- generates consecutive elements starting with the **first** (a number), continuing with all values obtained by subtracting **step** (a number, different from zero) to the previous one until the **limit** is reached (exclusively), i.e. the last element must be **greater** than limit.

- The following iterators are equivalent

```
V = [5,3,1,-1]
range(5,-2,-1)
```

- And the following ranges are empty

```
range(5,6,-1)
range(-5,-2,-2)
```

Iterators

- Iterators can also be used to initialise vectors and matrices.

```
In : L = [0 for i in range(3)]
In : L
Out: [0,0,0]
In : M = [[0 for i in range(2)] for j in range(3)]
In : M
Out: [[0,0,0],[0,0,0]]
In : M = [[i for i in range(2)] for j in range(3)]
Out: [[0,1,2],[0,1,2]]
```

Iterative Execution - FOR

- Back to the FOR statement.
- The following functions compute the same result from a vector passed as an argument.

```
def name_1 (V):
   """ returns ??? """
   s = 0
   for v in V:
      s = s + v
   return s
```

```
def name_2 (V):
   """ returns ??? """
   s = 0
   for i in range(len(V)):
      s = s + V[i]
   return s
```

- What do they compute, for example with V = [1,3,5,7]) ?
  - And in general?

Iterative Execution - FOR

- The previous functions use variable s as an **accumulator**.
  - At each iteration the accumulator is updated to take into account the elements of the vector already considered.
  - The update of the accumulator variable can be viewed in “debugging” mode, i.e. printing the values to be observed when they are updated.

```
In : V =[ 2 6 1 7]
Out: x = name_1(Z)
 0
 2   # 0 + 2
 8   # 2 + 6
 9   # 8 + 1
 16  # 9 + 7
In : x
Out: 16
```

```
def name_1 (V):
   """ returns ??? """
   s = 0
   print(s)
   for v in V:
      s = s + v
      print(s)
   return s
```

Iterative Execution - FOR

- The following examples uses the same technique, but include an if statement inside the for, so that only some elements produce changes to the accumulator variable.
- What do these functions compute?

```
def name_3(V):
    """ returns ??? """
    x = -m.inf
    for v in V:
        if v > x:
            x = v
    return x
```

```
def name_4(V):
    """ returns ??? """
    x = +m.inf
    for i in range(len(V)):
        if V[i] < x:
            x = V[i]
    return x
```

- Note that this technique can be used
  - with any operation that is commutative and associative, as is the case of operations **sum, product, max** and **min**; and
  - The accumulator is initialized with the neutral element of the operation (**0** for **sum**, **1** for **product**, **-inf** for **max** and **+inf** for **min**)

Iterative Execution - FOR

- Again the behaviour of these functions can be “debugged”.

```
def name_3(V):
   """ returns ??? """
   x = -m.inf
   print(x)
   for v in V:
      if v > x:
          x = v
      print(x)
   return x
```

```
In : Z =[2 6 1 7]
In : x = name_3(Z)
-inf
2
6
6
7
In : x
7
```

```
def name_4(V):
   """ returns ??? """
   x = +m.inf
   print(x)
   for i in range(len(V):
      if V[i] < x:
         x = V[i]
      print(v)
   return x
```

```
In : Z =[2 6 1 7]
In : x = name_3(Z)
+inf
2
2
1
1
In : x
1
```

Nested FORs

- When dealing with matrices it is usual to adopt two iterative variables to represent the indices of the rows and columns of the matrix.
- This is illustrated in the following example, taking a matrix as an argument.
- What does it compute?

```
def name_5(M):
   """ returns ??? """”
   s = 0;
   # print(s)
   for i in range(len(M)):
      for j in range(len(M[i]):
         s = s + M[i][j]
         # print(s)
   return s
```

Nested FORs

- Again the behaviour of this function may be debugged:

```
In : M =[[2,6,3],[1,0,8]]
In : x = name_5(M)
0
2    #  0 + M[0][0]
8    #  2 + M[0][1]
11   #  8 + M[0][2]
12   # 11 + M[1][0]
12   # 12 + M[1][1]
20   # 12 + M[1][2]
In : x
Out: 20
```

```
def name_5(M):
    """ returns ??? """
    s = 0;
    # print(s)
    for i in range(len(M)):
        for j in range(len(M[i]):
            s = s + M[i][j]
            # print(s)
    return s
```

Nested FORs

- Actually the same result could be obtained by summing the elements of the matrix by columns:

```
In : M =[[2,6,3],[1,0,8]]
In : x = name_6(M)
0
2    #  0 + M[0][0]
3    #  2 + M[1][0]
9    #  8 + M[0][1]
9    # 11 + M[1][1]
12   # 12 + M[0][2]
20   # 12 + M[1][2]
In : x
Out: 20
```

```
def name_5(M):
    """ returns ??? """
    s = 0;
    # print(s)
    for j in range(len(M[0])):
        for i in range(len(M)):
            s = s + M[i][j]
            # print(s)
    return s
```