More on Functions; WHILE instructions

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais

1st Semestre 2020/2021

Iterative Execution - WHILE

- In many cases, although a block of instructions is to be repeated, it is not known before hand how many times it should be iterated.
- For example, to find an element in a vector (or a word in a sequence of text), one might not have to look at **all** the elements of the array/matrix/text, since the element may be found before. In this case, the use of a FOR instruction (although possible) might not be desirable.
- In these cases, the WHILE instruction should be used. The WHILE instruction has the following syntax in Python

```
while <CONDITION>:
    WHILE-BLOCK
```

Iterative Execution - WHILE

```
while <CONDITION>:
    WHILE-BLOCK
```

- The behaviour of this instruction is quite intuitive. When the program reaches this instruction
  1. The CONDITION is assessed
  2. If the condition is not satisfied the WHILE-BLOCK is not executed and the program “jumps” to the next instruction.
  3. Otherwise, the WHILE-BLOCK is executed.
  4. After executing the block, the program goes back to step 1 (to assess the CONDITON again, …).
- **NOTE**: Care has to be taken in the specification of the condition and the WHILE-BLOCK. In particular, if this block does not change the variables involved in the CONDITION, so as to make it eventually false, the program **loops forever**!

Euclid’s Algorithm

- This instruction is illustrated with the **Euclid's algorithm** that finds the greatest common divider of two integers, with the following algorithm.

1. Take the two numbers, and make them A and B, ensuring that A is no less than B.
2. While A is greater than B
   - Obtain C, the difference between A and B (i.e. C = A – B);
   - Rename the numbers B and C, such that A becomes the larger of them and B the smallest.
   - Check again the condition and iterate as many times as needed.

- When A becomes equal to B, the iterations stop.
- The GCD of the initial numbers is A.

Euclid's Algorithm

Example:

- Let the numbers be 270 and 72, and see the evolution of the values of **a**, **b** and **c**.

a	b	c = a-b
270	72	198
198	72	126
126	72	54
72	54	18
54	18	36
36	18	18
18	18	0

- Hence 18 is the GCD of 270 and 72.

Euclid's Algorithm - WHILE

- The Euclid's Algorithm can be implemented with the following function:

```python
def euclid(p, q):
    """ computes m, the greatest common divider of p and q."""
    a = max(p,q)
    b = min([p,q])
    while a > b:
        c = a - b
        if c < b:
            a = b    # the order between a and b
            b = c    # cannot change, i.e. a >= b
        else:
            a = c    # and b remains b
        # print("a =", a,"; b =", b)
    return a         # since it is not a > b, then a = b
```

Euclid’s Algorithm - WHILE

- A trace of the function execution shows how the values of f2, f1 and f are maintained

```
...
while a > b:
   c = a – b
   if c < b:
      a = b
      b = c
   else:
      a = c
   # print("a =", a,"; b =", b)
...
```

```
In : m = euclid(270, 72)
a = 198 ; b = 72
a = 126 ; b = 72
a = 72 ; b = 54
a = 54 ; b = 18
a = 36 ; b = 18
a = 18 ; b = 18
In : m
Out: 18
```

Iterative Execution - WHILE

- We can go back to the problem referred above of finding a value in a vector.
- In particular we are interested in specifying a function **find/2** that takes
  - A number as the first argument; and
  - A list (vector) as the second argument;

  and returns
  - The index of the first position in the list where that element appears.
- **Note**: If there is no such element the function should return None.
- Some examples:
  - `find(3, [5, 8, 4, 3, 6, 8, 2])` → `3`
  - `find(8, [5, 8, 4, 3, 6, 8, 2])` → `1`
  - `find(9, [5, 8, 4, 3, 6, 8, 2])` → `None`

Iterative Execution - WHILE

- Before implementing the function we should design a convenient algorithm to solve this problem. Informally
  - While you have not found it and there is a next element
    - Look at the next element of the array to see if it is the intended one
  - Report the index of the element where you found it
- Although the skeleton of the algorithm is there, a few points must be taken care
  1. Where do we start from
  2. What if the element is not in the array

- Firstly, we must guarantee that we look at the first element, … if there is one!
- Secondly, if there are no more elements to look at, the algorithm must return None.
- These issues may be dealt with in the specification of the **find/2** function

Iterative Execution - WHILE

- The algorithm can now be implemented as function find/2, shown below

```
def find(x, V):
    """this function returns k, the first position, where
    v is in array V. It returns None if v is not present."""
    i = 0              # start searching at position i = 0
    n = len(V)
    while i < n and V[i] != x:   # while it is worth looking
        i = i + 1
    if i < n:                    # x was found in position i
        return i
    else:                        # x was not found
        return None
```

Iterative Execution - WHILE

- A last note on the condition that could have been used in the WHILE

```
    while i < n and V[i] != x:
```

- As we know, trying to read an element of an array past its size reports an error

```
In : A = [4, 7, 5]
In : A[4]
     IndexError: list index out of range
```

- Hence it is important that testing the value of the element in a certain index is only done after being sure that such index is within the bounds of the vector.

- Python short circuits the evaluation of Boolean expressions such as A and B (A or B):

1. Firstly, the Boolean expression A is assessed;
2. If A is False (resp. True) the condition is False (resp. True) and B is **not assessed**!
3. Otherwise B is assessed.
4. The value of the condition is the value of B.

WHILE vs. FOR

- When it is known the maximum number of times a cycle might be repeated, an instruction FOR might be used to force up to this (max) number of cycles
- In this case, when the condition to stop the cycle becomes True (i.e. the value was found), then the cycle should be interrupted and the index returned
- If the condition is never met, then None is returned.
- In the context of a function, the interruption is achieved with instruction **return**, (as below) that immediately ends the function execution.

```
def find_2(x, V):
    """this function returns k, the first position, where
    v is in array V. It returns None if v is not present."""
    n = len(V)
    for i in range(0,n):     # search indices i: 0 <= i < n
        if x == V[i]:        # if the element is found in position i
            return i         # return the value of i
    return None              # if x is not found return None
```

Nested Functions

- As functions become more complex, their design relies on other functions, either system defined functions or user functions previously defined.
- For example if the **sin/1** function has been defined (in **`library math as m`**) then function **tg/1** could have been defined in the obvious way (with the same meaning of function **m.tan**)

```
def tg(x):
    """this function returns the tangent of angle x,
    computed from the sin of that angle"""
    s = m.sin(x)
    c = sqrt(1-s**2)
    if c != 0
        t = s/c;
    else:
        t = m.inf
    return t
```

- As we already knew, functions can call **other** functions. Assuming the called functions terminate, the calling functions will also terminate.
- However, what happens when a **function calls itself**?

Recursive Functions: Factorial

- When functions call themselves, i.e. they are defined **recursively**, one must be careful so as to guarantee that they do terminate.
- Take for example the case of the function **fact/1** defined recursively to obtain the factorial of a non-negative integer, i.e. the same as function factorial, pre-defined in Python library math.
- This functionality can of course be defined **iteratively**, by means of the **accumulation** technique seen in the previous lecture, implemented with a for loop.

```
def fact_ite(n):
    """this function computes iteratively the factorial of n"""
    f = 1
    for i in range(1,n+1): # i varies from 1 to n
        f = f * i
    return f
```

Recursive Functions: Factorial

- A more “mathematical” definition could however be used to guide the function implementation:

$$n! = \begin{cases} 1 & \text{if } n \le 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases}$$

```
def fact_rec(n):
    """this function computes recursively the factorial of n"""
    if n <= 1:
        return 1
    else:
        return n * fact_rec(n-1)
```

- Notice that in the implementation of this recursive function, the termination condition must be tested **before** the recursive call is made.

- Otherwise the program **loops forever**!

Recursive Functions: Factorial

- In fact, Python avoids infinite recursion, by setting a limit on the number of recursive call that are made.
- The current recursive limit is obtained by method sys.getrecursionlimit().
- This limit may be changed to k, with method sys.setrecursionlimit(k)

```
In : import sys
In : sys.getrecursionlimit()
Out: 3000
In : z.fact_rec(30)
Out: 265252859812191058636308480000000
In : sys.setrecursionlimit(55)
In : sys.getrecursionlimit()
Out: 55
In : z.fact_rec(30)
......
RecursionError: maximum recursion depth exceeded in comparison
```

- Note: the recursion limit is not exactly the number of recursive calls.

Recursive Functions: Greatest Common Divider

- The same recursive technique may be used to define the GCD of two numbers, taking into account that :

$$\texttt{gcd(m,n)} = \begin{cases} \texttt{m} & \texttt{if m = n} \\ \texttt{gcd(min(m,n),abs(m-n))} & \texttt{if m} \neq \texttt{n} \end{cases}$$

```
def gcd(p, q):
    """ computes m, the greatest common divider
    divider of p and q."""
    if p == q:
        return p
    else:
      a = min(p,q)
      b = abs(p-q)
      return gcd(a, b)
```

- Note again that in this recursive function, the termination condition is tested **before** the recursive call is made

Doubly Recursive Functions: Fibonacci Numbers

- A final example of a function that might be defined recursively returns the n^{th} Fibonacci element of the series

`1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …`

- Note that in this series, every element is the sum of the two previous elements.
- Hence the function can be defined recursively as

```
fib(n) =  { 1                        if n <= 2
          { fib(n-1)+ fib(n-2)       if n > 2
```

- There is a (significant) difference in this case, which is the fact that the function is recursively called twice, as we will analyse later.
- But from a modelling point of view, the recursively defined function can be implemented as before.

Doubly Recursive Functions: Fibonacci Numbers

$$
\texttt{fib(n)} = \begin{cases} \texttt{1} & \texttt{if n <= 2} \\ \texttt{fib(n-1)+ fib(n-2)} & \texttt{if n > 2} \end{cases}
$$

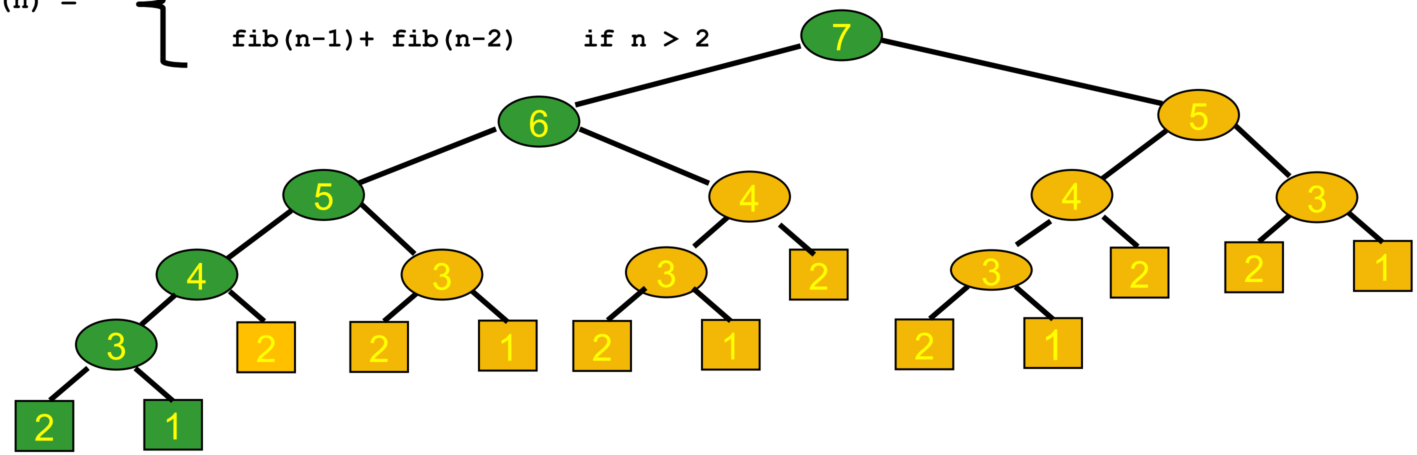
```
def fib_rec(n):
    """ This function computes (doubly) recursively the
    nth number of Fibonacci"""
    if n <= 2:
        return 1
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

- Although the termination condition is tested **before** the recursive calls are made, now there are two recursive calls and this has a big impact on the execution
- In particular, many instances of function fib, *with the same input arguments*, are called several times, in fact an **exponential** number of times!

Doubly Recursive Functions: Fibonacci Numbers

- In fact, we can trace the computation, and see that the following calls are made

```
fib(n) = { 1                     if n <= 2
         { fib(n-1)+ fib(n-2)    if n > 2
```



- fib(7) is called 1 time
- fib(6) is called 1 times
- fib(5) is called 2 times
- fib(4) is called 3 times
- fib(3) is called 5 times

- In general,
  - fib(3) is called fib(n-2) times
  - fib(4) is called fib(n-3) times, …
- and fib(n) grows exponentially!

```
1, 1, 2, 3, 5, 8,13, 21, 34, 55, 89, 144, 233,
377, 610, 987, 1597, 2584, 4181, 6765, 10946, …
```

Double Recursive Functions: Fibonacci Numbers

- To avoid this exponential explosion with double recursive functions one should resource to an iterative version of the algorithm, that although less “elegant” is much more efficient.

- The iterative version, shown below, maintains the previous 2 fibonacci numbers in two variables f2 and f1 that are added to obtain the current fibonacci number.

```
def fib_ite(n):
    """ This function computes iteratively the
    nth number of Fibonacci""""
    f = 1
    f2 = 1
    f1 = 1
    for i in range(3,n+ 1):  # i ranges from 3 to n
        f = f2 + f1
        print("f2 = ", f2, " + f1  = ", f1 , " ->  f = ", f )
        f2 = f1
        f1 = f
    return f
```

- Note that the iterations only take place for i ≥ 3, and stop for i = n

Double Recursive Functions: Fibonacci Numbers

- A trace of the function execution shows how the values of f2, f1 and f are maintained

```
    for i in range(3,n+ 1):  # i ranges from 3 to n
        f = f2 + f1
        print("i = ", i, " : f2 =", f2, "+ f1 = ", f1 , "-> f = ", f )
        f2 = f1
        f1 = f
    return f
```

```
In : fib_ite(8)
i =  3 : f2 = 1 + f1 = 1 -> f =  2
i =  4 : f2 = 1 + f1 = 2 -> f =  3
i =  5 : f2 = 2 + f1 = 3 -> f =  5
i =  6 : f2 = 3 + f1 = 5 -> f =  8
i =  7 : f2 = 5 + f1 = 8 -> f =  13
i =  8 : f2 = 8 + f1 = 13 -> f =  21
Out: 21
```