WHILE instruction; Strings

Pedro Barahona
DI/FCT/UNL
Computational Methods
1st Semester 2021/22

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:

```
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.

```python
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:

```
         ┌  1              if n <= 1
n! =    ─┤
         └  n * (n-1)!     if n  > 1
```

```
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

Text Processing

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- Much useful information is not numeric and takes the form of text (e.g. names, documents, ...). Hence the need to represent text and to subsequently process it.
- All programming languages support text data types, namely
  - **Characters**; and
  - **Strings** (sequences of characters).
- Basic 128 characters, include letters, digits, punctuation and control characters, and are usually represented by their **ASCII** (American Standard Code for Information Interchange) codes.
- Notice that 128 different characters require 7 bits to be represented (128 = 27).
- With an 8th bit (initially meant for parity checking), the extended ASCII code allows the representation of 128 more characters used in several languages (other than English).

Text Processing

- The characters represented in 7bit ASCII code are:
  - Letters (52), uppercase (26) e lowercase (26)
  - Digits (10)
  - Space and other punctuation “visible” characters (34)
    - ‘ “ ( ) [ ] { } , . : ; = < > + - * \ | / ^ ~ ´ ` # $ % & _ ! ? @
  - Control (invisible) characters (32)
    - horizontal tab (\t), new line (\n), alert (\a), ...
- With an 8th bit, other 128 characters can be represented, such as
  - ç, ã, ñ, š , ø , ∞, ← φ, Σ, ш, غغ ,א ,ך
- The representation of other alphabets (Chinese, Arab, Indian, ...) require 16 bits (a total of 216 = 65536 characters) and is supported in Unicode (widely adopted in the Internet).
- Unicode (UTF) subsumes the ASCII code (the initial 256 characters are the same).

Strings

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- Strings are sequences of characters, and text can be regarded as a “big” string.
- To assign a variable with a string, the text must be delimited by quotation marks (") or single quotes ('). For example,
  - `x = "this is a string"`
- Having two delimiters is quite handy, when the text includes one of them, as in
  - `name = "Rui d' Almeida"` ; or
  - `next = 'He said "Enough" and left.'`

  … although escape sequences can be used
  - `name = 'Rui d\' Almeida'` ; or
  - `next = "He said \"Enough\" and left."`

  … and these are sometimes unescapable
  - `sentence = "Rui d' Almeida said \"Enough\" and left."`
  - `sentence = 'Rui d\' Almeida said "Enough" and left.'`

Escape Sequences

- The following escape sequences are useful for referring special non visible characters, namely control characters.
- There are some differences in the handling of the delimiters and escape characters, and the “” delimiter should be preferred. The following escape sequences are accepted in Python (e.g. in a print statement).

`\\`	**back slash**	**(\)**	
`\"`	**quotation**	**(”)**	
`\'`	**single quote**	**(‘)**	
`\0`	nil	(code 0)	
`\a`	alert	(code 7)	
`\b`	back	(code 8)	– overwrites previous character
`\f`	new page	(code 12).	
`\n`	**new line**	**(code 10).**	
`\r`	return	(code 13)	– overwrites previous line
`\t`	**horizontal tab**	**(code 9).**	
`\v`	vertical tab	(code 11).	

String Operations

- Strings are encoded as lists of characters of characters, so the usual operations on vectors can be used to compose and decompose strings.

Concatenation

- Strings can be concatenated with the + operator, as with lists.

```
In : v1 = [1,2,3]
In : v2 = [4,5,6]
In : v1 + v2
Out: [1,2,3,4,5,6]
In : name = "Rui"
In : surname = "Santos"
In : full = name + surname
In : full
Out: "RuiSantos"
In : full = name + " " + surname
Out: "Rui Santos"
```

String Operations

Projection (Extraction) of Substrings

- Projection of strings to some of their substrings (or characters) can be obtained through the usual vector operations

```
In : text = "This is a string."
In : text
Out: 'This is a string.'
In : text[0:4].          # all chars between the 1st and 5th
Out: 'This'
In : text[-7:-1]
Out: ‘string'
```

- Several methods are defined in the class string (cf. the dir function)

```
In : dir(text)
Out:
['__add__’,
 ...
'zfill']
```

String Operations

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Substring Search

- If one is interested in finding the (first) position(s) where a substring occurs within a string, the **find** and **rfind** methods can be used.

```
In : text = 'This is a string.'
In : text.find('string')
Out: 10
In : text.find('i')
Out: 3
In : text.rfind('i')
Out: 13
In : text.find('z')
Out: -1   # not found
```

String Operations

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Splitting Strings

- In many cases we are interested in splitting a string by some character(s) that is used as a separator (for example a semi-colon (;), a tab (‘\t) or a space.
- Method split() returns a list of strings, without the separators

```
In : line = 'abd; def; 123'
In : line.split(';')
Out: ['abd', ' def', ' 123']
In : line = ‘12\t24\t45.8\n’
In : line.split(‘\t’)
Out: ['12', '24', '45.8\n']
```

- Note: Beware of spaces and “end of line” (‘\n’) characters that might be maintained in the individual strings.

String Operations

“Cleaning” Strings

- In many cases we are not interested in leading and trailing spaces, as well as white characters such as tabs and end-of-lines (e.g. when they are read from files).
- They can be eliminated with methods **strip**.

```
In : line = "   This is a line.  \n"
In : len(line)
Out: 21
In : line.strip()
Out: 'This is a line.’
In : len(line.strip())
Out: 15
```

String Operations

Comparing Strings

- Strings may also be compared lexicographically (i.e. alphabetically).
- Notice that lower and upper cases are different (in ASCII, upper cases are before lower cases).

```
In : "abc" == "abc"
Out: True
In : "abc" > "abd"
Out: False
In : "A" < "a"
Out: True
In : "A" < "5"
Out: False
In : "5" < 5
TypeError: '<' not supported between instances of 'str' and 'int'
```

String Types

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Strings and Numbers

- Strings are different from numbers, and different operations apply to these types.
- But converting strings to numbers and vice-versa is possible (but beware of different types of numbers).

```
In : '45'+'12'
Out: '4512'
In : '45'*'12'
TypeError: can't multiply sequence by non-int of type 'str'
In : int('45')
Out: 45
In : str(34)
Out: '34'
In : float('45.7')
Out: 45.7
In : int('45.7')
ValueError: invalid literal for int() with base 10: '45.7'
```

String Type Information

Information Functions about Types

- In addition to the conversion functions a number of methods are available to strings to obtain the types of characters, namely

- isalnum - string composed of alphanumeric characters
- isalpha - string composed of alphabetic characters
- isascii - string composed of ASCII characters (7 bits, no special characters)
- isdigit - string where all characters are digits
- isidentifier - string is a valid identifier
- islower - string where all characters are lower case letters
- isprintable - string where all characters are printable (spaces, tabs, eol)
- isspace - string where all characters are non printable (spaces, tabs, eol)
- istitle - string starting with an upper case letter followed by lower case
- isupper - string where all characters are upper case letters

String Type Information

Some examples

```
In : 'ab5dc'.isalnum()
Out: True
In : 'ação'.isascii()
Out: False
In : '3456'.isdigit()
Out: True
In : '_45'.isidentifier()
Out: True
In : ‘a45'.isidentifier()
Out: True
In : '56 67'.isprintable()
Out: True
In : '\t   \n'.isprintable()
Out: False
In :'Doutor'.istitle()
Out: True
```

```
In : 'ab5dc'.isalpha ()
Out: False
In : ‘facto'.isascii()
Out: True
In : ‘34a56'.isdigit()
Out: False
In : 'a.45'.isidentifier()
Out: False
In : '45a'.isidentifier()
Out: False
In : '56 67’.isspace()
Out: False
In : '\t   \n’.isspace()
Out: False
In :'DR.'.istitle()
Out: True
```