python™

Programming with Python for Data Science

Unit Topics

- Conditionals
- Loops and comprehensions
- Functions
- Modules

Learning objectives

- Write conditional operators in Python
- Describe the available Python loops
- Know how to write a function with input parameter, and how to modify a variable with a function
- Know how to load a set of functions

if/elif/else

In computer programming, conditional statements are used to perform different computations or actions depending on whether the condition is True or False.

```
if   <some condition>:
     <TRUE code>
else:
     <FALSE code>
```

The keyword **`if`** marks the beginning of the decision-making process
A colon **`:`** follows the condition that must be met
The keyword **`elif <other condition>:`** is **optional** and marks the beginning of another condition check. Note that the first occurrence of True stops the other conditions evaluation.
The keyword **`else:`** is **optional** and marks the beginning of the last alternative.

The code that processes data must be indented under the lines of the keywords.

and/or

1. **`X or Y`**
   If X is `False`, returns Y (that can be `True` or `False`), else returns X
   (`if x then True else y`)

2. **`X and Y`**
   If X is `False`, returns X, else Y
   (`if x then y else False`)

3. **`not X`**
   If X is `False`, returns `True`, else `False`

- As you can see the **and** and **or** operators are "**short circuits**":
  - **`or`** **evaluates** the second argument **only if** the first is `False`
  - **`and`** **evaluates** the second argument **only if** the first is `True`

- `not` has lower priority than non-boolean operators, so

  ```
  not a==b
  ```

  is interpreted as

  ```
  not (a==b)
  ```

Sample program

Write a program which asks the user to enter a positive integer 'n' (Assume that the user always enters a positive integer) and based on the following conditions, prints the appropriate results exactly as shown in the following format:

1. when 'n' is divisible by both 2 and 3 (for example 12), then your program should print the string BOTH
2. when 'n' is divisible by only one of the numbers (as 8 or 9), your program should print the string ONE
3. when 'n' is neither divisible by 2 nor divisible by 3 (as 25) your program should print NEITHER

```
user_response=input("Type a number:")
x = int(user_response)
if x % 2 == 0 and x % 3 == 0:
    print("BOTH")
elif (x % 2 != 0 and x % 3 == 0) or (x % 2 == 0 and x % 3 != 0):
    print ("ONE")
else:
    print("NEITHER")
```

While loop

The while loop repeatedly executes one or more statements, as long as a condition is true.

```
while <some condition>:
    statement(s)
    ...
else:
    final statements
```

In the field <some condition> you can put any expression which evaluates to either **True** or **False**
As we have seen for the if keywords, a colon is required, and then the statements to be executed must be written indented below the while line.

The else code block is **optional**, and is executed when the condition of the while loop becomes False, also for the first time (so unless there is a break in the loop it is always executed)

Remember that if the condition in front of the while keyword never becomes false, then the loop never ends.

For loop

The for loop allows you to iterate through a sequence, and execute a code for each item in that sequence.

```
for item in seq:
    statement(s)
    ...
else:
    final statements
```

The target identifier is not previously declared , and it is similar to any other name in Python; when the cycle goes on each object in the sequence is assigned to the target "item", one after the other.

The statements can be uncorrelated with the item of the sequence.

The else code block is **optional**, and is executed when the condition of the for loop becomes False, also for the first time (so unless there is a break in the loop it is always executed)

For loop

- The for loop is designed in Python to work on lists OR on any other object that Python considers **iterable**.

- **An iterable object is a container capable of returning its items one by one.**

  Objects of this type are all objects that have one of the following functions defined:
  `__iter()__`
  `__getitem()__`

- The latter can take sequential indexes starting from zero (and raises an IndexError when the indexes are no longer valid)

- When an iterable is passed to the BIF `iter()`, which happens automatically inside a for loop, it returns an ***iterator***.

- An *iterator* is an object that represents a data stream, and enables the programmer to traverse the container.

- In other words, in Python, an iterable is an object which can be converted to an iterator, which is then iterated through during the for loop; **this is done implicitly**

Examples

```
>>> seq = [1,2,3,4,5]
>>> for item in seq:
        print item


>>> i=1
>>> while i< 5:
        print('i is: {}'.format(i))
        i+=1

>>> i=1
>>> while i< 5:
        print('i is: %s' % i)
        i+=1

>>> for x in range(0,10):          #In 3.x range gives an immutable seq type
        print(x)

>>> for x in list(range(10)):
        print(x)
```

Mutable objects and loops

Let's try the following code segment:

```
>>> list = [3,4,2,1]
>>> for number in list:
        print('number {} in list {}'.format(number, list))
        if number > 2:
            list.remove(number)

number 3 in list [3, 4, 2, 1]
number 2 in list [4, 2, 1]
number 1 in list [4, 2, 1]
```

The problem is that in the second iteration, the for loop uses the index 1, e then skips the second list item, the “4”, that now has index 0

The message is simple:
never ever change a list (or any mutable object) on which you are iterating!

Sample code

Write the correct code that removes, from the list L=[9,8,1,2], every item greater than two with a for loop.

```
>>> L = [9,8,1,2]
>>> for number in L[:]:
        print('number {} in list {}'.format(number, L))
        if number > 2:
            L.remove(number)

number 9 in list [9, 8, 1, 2]
number 8 in list [8, 1, 2]
number 1 in list [1, 2]
number 2 in list [1, 2]
```

Write a while loop that removes from the list L=[9,8,1,2] every item greater than two

```
>>> L = [9,8,1,2]
>>> index=0
>>> while index <= (len(L)-1)
        if L[index]>2:
            L.remove(L[index])
        else:
            index=index+1              #the index is updated
print(L)
```

List comprehension

- The list comprehensions are a very common way to build new lists starting from existing lists.

```
newlist = [expression for name in list]
```

- `name` is an identifier that loops over each item of `list`; for each item `expression` is computed, and the resulting item is added to the new list, that is eventually returned to the identifier on the left

- `expression` **must** be a **valid** calculation for **every** item in `list`.

- If `list` contains other containers, then `name` can be a containers of names that must match the `list` items

Examples

```
>>> x = [1,2,3,4]
>>> out = []
>>> for num in x:
        out.append(num**2)



>>> x = [1,2,3,4]
>>> out = [num**2 for num in x]


>>> x = [1,2,3,4]
>>> out = [num**2 for num in x if num%2==0]   #Filtered list comprehension


>>> def subtract(a,b):
        return a-b
>>> list= [(6,3),(1,7),(5,5)]
>>> [subtract(y,x) for (x,y) in list]
[-3,6,0]

>>> li=[3,2,4,1]
>>> [elem*2 for elem in [item +1 for item in li]] #Nested (Danger!)
[8,6,10,4]
```

Functions

A **function** is a reusable block of code, which can also have input arguments.

You define a function using the keyword **def**, providing a name for the function and specifying a list, even empty, of input arguments in parentheses

```
def <function name>(<argument(s)>):
        <function code>
```

The parentheses are mandatory, unlike the arguments. The code of the function must be indented as usual. The parameters can have defaults.

The name uniquely identifies the function (The number, name or type of arguments can not be used to distinguish between two functions with the same name)

Scope:

1. If a variable is assigned inside a *def,* it is *local* to that function.
2. If a variable is assigned outside any *def,* it is *global* to the entire code.
3. Do NOT modify global variables inside a function!

A function can return an arbitrary number of items to the caller.

Examples

```
>>> def square(num):
        return num**2
>>> square()
TypeError: square() missing 1 required positional argument: 'num'
>>> out=square(4)
>>> out
16


>>> def square(num=2):
        """
        THIS IS a DOCUMENTATION STRING
        CAN GO MULTIPLE LINES
        THIS FUNCTION SQUARES A NUMBER.
        """
        print(num**2)
>>> square()
4
>>> square(5)
25
>>> square
<function __main__.square>
>>> help(square)
```

Examples

What do these codes print, and why?

```
>>> x = "UNIBA"
>>> def func():
        print x
>>> func()
```

UNIBA

Since the variable x is not assigned inside the function, it is considered global (you can use the global variables inside a function)

```
>>> x = "UNIBA"
>>> def func():
        x='INFN'
        print x
>>> func()
>>> print x
```

INFN

UNIBA

The assignment of x inside the function hides the global x, so the x='INFN' is local to the function and does not modify the global x

```
>>> x = "UNIBA"
>>> def func():
        global x
        x='INFN'
>>> func()
>>> print x
```

INFN

The `global` statement force the assignment in the function to use the identifier x in the global scope, that now can be modified.

Functions and input variables

C and C ++ use by default the "call by value" schema, by making a copy of the variables within the function, but you can use pointers and get the "call by reference".

This last behaviour is standard in Fortran.

So what happens in Python?

Functions and input variables

- Python functions argument passing is a sort of «**call by assignment**», in the sense that an assignment operator is applied between the argument and the variable used in the call.

- Python always passes a reference to an object. But: if the object is mutable you can modify it, otherwise you cannot.

- **Concretely:** the input function parameters are passed:
  1. with a «*call by reference*» (pointer), if the data type is mutable
  2. with a «*call by value*» (copy) if the data type is immutable.

- It is then a **good** practice to **always return** every parameter modified inside the function. Note that with a single return statement you can return any number of variables.

Examples

```
>>> def change1(some_list):
        some_list[1] = 4

>>> x = [1,2,3]
>>> change1(x)
>>> print x
```

What is the value of x? **x = [1,4,3]**

```
>>> def change2(x):
       x = 0

>>> y = 1
>>> change2(y)
>>> print y
```

What is the value of y? **y = 1**

Variable number of arguments

You can define functions with a variable number of arguments

```
def somefunc(a, b, *args):
    # args is a tuple of all supplied positional arguments
    ...
    for arg in args:
        <work with arg>
```

A double asterisk indicates a variable length set of **named** arguments (a dictionary), which must be passed to the function when it is called.

```
def somefunc(a, b, *args, **kwargs):
# args is a tuple of all supplied positional arguments
# kwargs is a dictionary of all supplied keyword arguments
...
for arg in args:
    print arg
for key in kwargs.keys():
    print key, ' : ', kwargs[key]
```

map and lambda functions

```
>>> def times2(var):
        return var*2

>>> times2(2)
4
```

Now I want to apply this function to the sequence `seq=[1,2,3,4,5]`

```
>>> map(times2,seq)
<map at 0x106a3f6a0>
>>> list(map(times2,seq))
[2, 4, 6, 8, 10]
```

But what if I do not really need a new function definition, but only a user defined transformation on the items of my sequence? It is possible to reduce the function definition removing the **name**, the **def** statement and the **return** keyword:

```
>>> lambda var: var*2
<function __main__.<lambda>>

>>> list(map(lambda var: var*2,seq))
[2, 4, 6, 8, 10]
```

filter (and reduce)

The built-in function `filter` has a similar structure to `map`, but instead of mapping a function to every element of a sequence, it filter OUT elements from a sequence.

```
>>> filter(lambda item: item%2 == 0,seq)
 <filter at 0x105316ac8>

>>> list(filter(lambda item: item%2 == 0,seq))
[2, 4]
```

In words:
`filter` builds a list formed from the items of the sequence for which the lambda expression (or the function) returns `True`
`map` builds a list formed from the values of the lambda expression (or the function) computed for each item of the sequence.

`reduce` returns a single value computed applying the function on the first two elements of the sequence, then on the pair formed by the first result and the third item, and so on…

```
>>> values=('abc','xy','abcdef','xyz')
>>> mapped=map(lambda x:len(x),values)
>>> max_len=reduce(lambda x,y: max(x,y),mapped)
```

tuple unpacking

```
>>> x=[(1,2),(3,4),(5,6)]
>>> for item in x:
        print(item)
(1,2)
(3,4)
(5,6)

>>> for (a,b) in x:
        print (a)
1
3
5

>>> for a,b in x:
        print(a)
        print(b)
1
2
3
4
5
6
```

Modules

You can use three different syntax to import a module:

1. **`import moduleName (as alias)`**

   Any function (or class) defined in the module file `moduleName.py` is imported and usable in the script, but you **must** prepend moduleName to the name of the function

   ```
   moduleName.square (34)
   ```

2. **`from moduleName import *`**

   Any function (or class) defined in the module file `moduleName.py` is imported and useable in the script with its name:

   ```
   square(34)
   ```

3. **`from moduleName import square`**

   Only the square function is imported from the moduleName, directly in the main namespace:

   ```
   square(34)
   ```

If you want to add your own module's directory to Python use:

```
>>> module_dir=os.path.join(os.environ['HOME'],'my','modules')
>>> sys.path.insert(0,module_dir)
```

Exercises

A word is palindrome if it can be read in both directions. More analitically a palindrome word has the same character at the beginning and the end, and the remaining characters palindrome

Write a simple function *is_palindrome* that accept a string as input argument, and returns true if it's palindrome, false otherwise.

```
bob
a nut for a jar of tuna
in girum imus nocte et consumimur igni
sator arepo tenet opera rotas
Never odd or even
Murder for a jar of red rum
Are we not drawn onward, we few,
    drawn onward to new era
Ma is as selfless as I am
Now I see bees I won
```

Exercises

The following code searches the value 2**X in a given list of numbers, and if founds it returns the index.

```
L=[1,2,4,8,16,32,64]
X=5
found=False
i=0
while not found and i<len(L):
    if 2**X ==L[i]:
        found=True
    else:
        i=i+1
if found:
    print ((2**X), 'found at index ', i)
else:
    print ((2**X), ' not found')
```

Rewrite it in a more Python like fashion, removing the loop.

Exercises

Write a code that prints every number from 100 to 1000 that is divisibile to 5 AND 3.

The code must print the numbers in lines of 10 numbers, with a white space between the numbers.

Exercises

The Ackermann function A(m,n) is defined by:

	= n+1	for m=0;
A(m,n)	=A(m-1,1)	for m>0 and n=0
	=A(m-1,A(m,n-1))	for m,n>0

Write a code that compute the Ackermann function, and use it to compute A(3,4) (The result must be =125)
Try to compute A(4,3)…

Exercises

Write the list comprehension that, from the lists

```
l1=[4,5,6,7,8]
l2=[8,9,10,11,12]
```

builds the matrix (list of lists) of the items products, then print it line by line:

```
4*8 4*9 4*10 ...
5*8 5*9 5*10 ...
...
```