Control flow tools

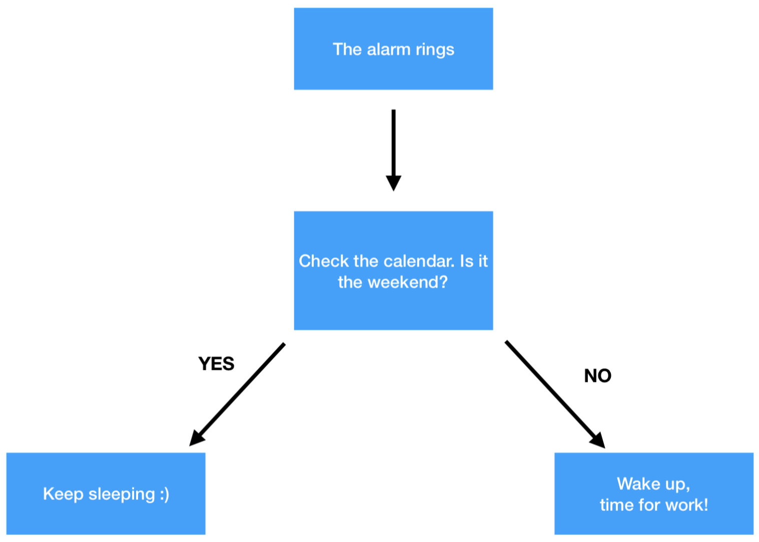
In this chapter, we are going to present a few tools that Python (but, in general, almost any programming language) provides us to control what the computer does. In a certain sense, if programming is a bit like cooking, the previous chapter presented the ingredients, whereas in this one we will focus on how to write the instructions (algorithm) to do that.

Conditional ("if") statements

When giving some instructions, we are often presented with multiple possible scenarios in front of us. Depending on which one we have, we would like to react by doing certain things, or others. If you think about it, you are presented with a situation of this kind every morning:

1. The alarm rings
2. Check the calendar
3. If it is Saturday, keep sleeping
4. If it is Sunday, go running
5. Else, wake up.

a useful, flow-chart representation of this type of "instruction" is given in the figure below.



In Python, a construct of this kind is called an "if statement" and has the following synthax:

```
if ( any expression which can be evaluated to True or False ):
    operation x
    operation y
    ...
elif ( some other expression which can be evaluated to True or False ):
    operation xx
    operation yy
    ...
else:
    operation xxx
    operation yyy
    ...
```

with this construct, the Python interpreter will evaluate the first expression after the `if` and, in case it is `True`, it will carry out operations x and y. If not, it will then check the second expression right after the `elif` command and again if true carry out operations xx and yy, otherwise it will just do the operations xxx and yyy.

Keep in mind that in general you can put any number of operation in any **indented** block (not just two like in the example above).

You should also be **very careful** about a few important things related to the syntax of an `if` statement:

- 1) Within the two brakets () you can put *any Python statement that returns a value of either `True` (or 1) or `False` (or 0)*.

So a valid statement might be `a > 1`, or `a == b` (if you have declared the value of the variables `a` and `b` before), so that Python knows how to check their value. Expressions of this kind are called boolean expressions (https://en.wikipedia.org/wiki/Boolean).

You can build arbitrary complex expressions by using the so-called boolean operators `and`, `or` or `not`, together with the operators `==` (the comparison operator "equal to", we have seen it before) and `!=` (the comparison operator "not equal to"), which should all be familiar to you from elementary mathematics. If not, have a look at this document (https://library.alliant.edu/screens/boolean.pdf).

- 2) In Python, you can literally write `and` (or the symbol `&`), `or` (or the symbol `|`) or `not`. For example, a valid Python expression is:

  ```
  a and b or not c
  ```

or, equivalently:

  ```
  a & b | not c
  ```

- 3) The third important thing to notice, and this of absolute importance in Python, is that *all the operations indented with respect to the lines starting with an `if`, `elif` (if a conditions is evaluated to true) or `else` (if none of the previous conditions was true) command will be executed*.

After a condition is satisfied, you enter in the corresponding indented block and Python will do all the idented operations, after which the interpreter will move immediately to the first **non-indented** line and keep reading from there. In other words, it will only read a single indented block and skip all others in the same construct. Let us make a (hopefully clarifying!) example. If you write:

```
a = 0
b = -1
if a > b:
    print( "Today is Monday" )
elif b == -1:
    print( "We have a lab" )
print( "Today is a nice day" )
```

when executed would only print out:

`Today is Friday` and `Today is a nice day` but **not** also `We have a lab`, although the condition `b==-1` is clearly satisfied.

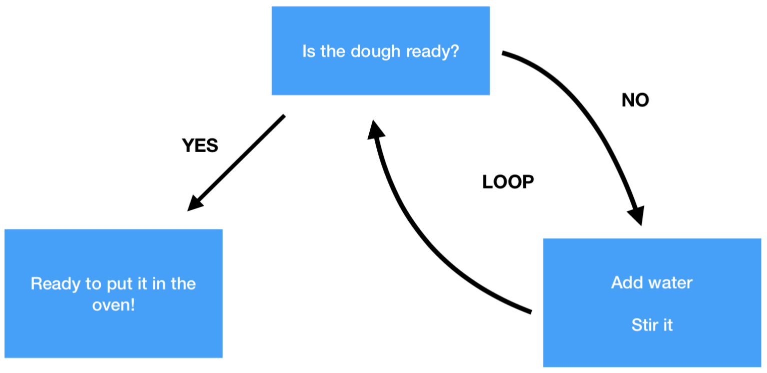
- 4) Finally, you should notice that in general whereas the `if` part of the construct is necessary, the `elif` and `else` parts are optional. It will depend on your needs whether or not you will want to have them.

Repeating instructions multiple times: "Loops'"

Computers are good at doing very repetitive things very fast. So fast that you can actually make them repeat calculations that would require you million of hours, while computers would execute them in the blink of an eye. Coding these operations requires the use of constructs called "loops".

In practice, a loop is the way we communicate to the computer to repeat (as in a loop) a certain operation until "something" happens. For example, because a certain condition is met, or because the computer has finished dealing with all the cases required. Exactly these two cases correspond to what are called `while` and `for` loops, as we are about to describe.

The `while` loop



Imagine you want to bake some bread. For this, you need to make the dough by yourself, without knowing how much water you need to add to flour to do that. What you could do to solve this dilemma is repeating the following instructions:

```
while ( the dough has not reached the right consistence): \\
    add a spoon of water
    stir it
```

In practice, you repeat a certain set of instructions **until a condition is met**. The general syntax for a while loop is the following:

```
while ( any valid boolean expression ):
    operation x
    operation y
    ...
```

As in the case of the `if` statement, *all operations indented with respect to the `while` command are repeated*, if the condition evaluates to `True`. After those operations are performed, the expression whithin brakets () is evaluated again. If its value is again `True`, the series of indented operations are repeated again, otherwise the program exits from the loop and execute the first non-indented operation after it.

Usually, the repeated operations inside the loop modify the value of some of the quantities inside the expression to be evaluated in parenthesis, so that at some point the program exits the loop. If this does not happen, you can get stuck infinitely inside it, and so does Python! Let us make an example:

```
i = 1
while i <= 10:
    print( i**2 )
    i += 1  # Quick note: i += 1 is equivalent to writing i = i + 1
print( 'End of the loop' )
```

prints the first ten squares (1, 4, 9, 16, ...100) and then exit the loop, printing 'End of the loop'. Instead, the following program would get you stuck inside the loop indefinitely (and keep printing the squares of higher and higher numbers), because the condition inside brakets is never met!

```
i = 1
a = 1
while a <= 10:
    print( i**2 )
    i += 1  # Quick note: i += 1 is equivalent to writing i = i + 1
print( 'End of the loop' )
```

The `for` loop

Imagine in a near future robots finally work for us. You have a list of friends and you want to give each of them a present. The set of instruction / algorithm for your robot could be, schematically:

```
for friend in [ a list of friends ]:
    think about a present
    buy the present
    wrap the present
```

A Python `for` loop does just that: it repeats a series of operations for every element inside some generic list (...or set or dictionary or tuple, actually!). In fact, the syntax is exactly:

```
for counter in [ a list of elements ]:
    operation x
    operation y
    ...
```

In this declaration, `counter` is a variable whose value automatically keeps changing at each iteration of the loop. Starting from the first element in the list, at each cycle it takes sequentially all the values of the elements inside the list. When `counter` has taken all the possible values in the list, the loop terminates and again *the first non-indented operation* is executed.

As for the case of the `while` loop, here as well for each iteration *all operations indented with respect to the line starting with the* `for` *declaration are repeated*. Let us illustrate this with an example:

```
myList = [ 1, 2, 3, 4 ,5 ]
for i in myList:
    a = i**3
    print( i )
```

would print sequentially the numbers 1,8,27,64,125, that is, the cube of the elements inside `myList` .

It is important to notice that Python can execute a `for` loop not only over lists, but also over tuples, sets and dictionaries as well! For example, one could write:

```
a = ( "a", "b", 3 )
for i in a:
    print(i)
```

which returns `a` , `b` and `3` . Try to experiment with it in the exercises provided for this part of the course.

The `range` function

With `for` loops, one Python function that comes very handy is the so called `range` function (the exact way in which functions work will be discussed in the next lecture but for now just assume functions as in their standard mathematical definition).

The syntax of the `range` function is the following:

```
range( first, last, step )
```

returns a list containing all integers starting from *first (included) until last (excluded), increasing each time by step*. Note that `first` , `last` and `step` can be any integer number, or any variable whose value is an integer. An example again might be easier to understand here:

```
range( 1, 11, 2 )
```

The code above returns the list `[ 1, 3, 5, 7, 9 ]` , i.e. the numbers from 1 to 11 (excluded), separated by steps of 2. We could have written in an equivalent way:

```
a = 1
b = 2
range( a, 11, b )
```

to obtain exactly the same series.

Note that in general `range` does not need to have as an input all three values `first` , `last` and `step` . If only two are given, Python assumes the first element provided is `first` , the second is `last` and step assumes the default value of 1. If only one value is given, that is interpreted as the value of `last` , whereas `first` takes the default value of 0 and again `step` takes the default value of 1. So again to make two simple examples,

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

returns the list `[ 1, 2, 3, 4 ]`

whereas if we wrote:

```
range( 7 )
```

we would obtain the list `[ 0, 1, 2, 3, 4, 5, 6 ]` .

A little side note here: We have loosely talked about `list` in this subsection on the `range` function, but `range` does not returns exactly a Python list. Instead, it returns a "generator", that is, an instruction to generate that specific sequence of number. This is something new introduced in Python 3 to make the language more efficient and, as long as range is used within a loop, there is no confusion in thinking that range returns exactly a list of elements. You can find the details here (https://www.geeksforgeeks.org/range-vs-xrange-python/).

Quick declaration of lists via a `for` loop

Here I would like to quickly discuss a quick and very powerful trick to declare lists, using the `for` loop inside them. The general syntax in this case is the following:

```
myList = [ f( i ) for i in [some  list] ]
```

where f(i) is any possible function of i and [some list] is any list of values. Maybe an example could be useful here. Let us imagine we want to have a list containing the cubic roots of the first 100 integers. In principle, we could do that using a `for` loop and the `range` function in the following way:

```
listOfRoots = [   ] # Line 1
for i in range( 1, 101 ): # Line 2
    a = i**( 1.0 / 3.0 )  # Line 3
    listOfRoots.append( a ) # Line 4
```

so we basically declare the empty list `listOfRoots` in line 1, we then start a loop in line 2, we generate the cubic root of i in line 3 (remember: $i^{1/n}$ is the n-th root of i!) and then in line 4 we append this value to the list. This program would perfectly work (try in your Jupyter Notebook to believe!).
However, we could more simply and more compactly use the following one-line command:

```
listOfRoots = [ i**( 1.0 / 3.0 ) for i in range( 1, 101 ) ]
```

This method of generating list is extremely useful, I strongly suggest you to learn it!

Controlling loop executions: the `continue` and `break` commands

When looping over a list, whether within a `while` or `for` loop, for certain iterations we might need to skip the loop completely, or just execute a part of the operations indented under it. We can do this by using the `continue` command.

Just to make an example, we might need to print out all names in a list, except for a few of them. The following examples shows how to do that:

```
myListNames = [ 'Mark', 'Gerbrand', 'Alex', `Paula', `Lisa' ]

for name in myListNames:
    if name == "Alex" or name == "Lisa":
        continue
    else:
        print( name )
```

would loop over the names in the `myListNames` and print them, **except when the counter `i` takes the values `Alex` or `Lisa`**. In practice, the general syntax is the following:

```
for counter in (any sequence of elements):
    if (any boolean expression that evaluates to True):
        continue
    else:
        operation x
        operation y
        ...
```

What happens is that at each iteration the expression inside the parentheses after the `if` command is evaluated and, if it is true, the Python interpreter simply go to the next iteration of the loop, without executing any command after it.

We have seen above the use of the `continue` command, which we can use when we do not want to execute the commands in the loop, but still remain inside it. In other cases, we would like not just to skip the operations in the loop, but to get out of it if a certain condition is met. This can be done using the `break` command.

As an example, let us imagine we want to print out all the squares of integer numbers less than 20, but stop if the square itself is bigger or equal to 100. We could do this by writing the following:

```
i = 1
while i < 20:
    a = i**2
    if a >= 100:
        break
    else:
        print( a )
    i += 1
```

which will indeed print out the numbers 1, 4, 9, 16, 25, 36, 49, 64, 81, which corresponds to the squares of the integers from 1 to 9, but **not** the square of 10 because that is equal to 100, at which point the Python interpreter execute the `break` command and exits the loop. In general, the syntax for a `break` command is the following:

```
while (some boolean expression equal to True):
    operation x
    operation y
    ...
    if (some other boolean expression equal to True):
        break
    else:
        operation xx
        operation yy
        ...
```

and the same synthax for the `for` loop. Basically, whenever for any reason once inside a loop Python reads a `break` command, the program exits the loop and goes to the first indented line *after the last* `while` *or* `for` *command*.

Finally, a little clarification is necessary here. Although I have showed the use of `break` and `continue` using a complete `if / else` construct to make more explicit which operations are **not** done if the `break/continue` commands are reached, the truth is that the `else` statement is not necessary. In fact, exactly the same effect would be obtained if, after the `break` or `continue` , all other commands to be normally repeated in the loop were simply indented outwards, at the same level of the `if` .

An example should make it crystal clear. The loop:\

```
while (some boolean expression equal to True):
    some list of operations
    if (some other boolean expression equal to True):
        break / continue
    else:
        some other list of operations
```

can be replaced by the **exactly equivalent** construct:

```
while (some boolean expression equal to True):
    some list of operations
    if (some other boolean expression equal to True):
        break/continue
    some other list of operations
```