6 Controlling programme execution - the 'if' statement

In the note books you have used so far you have only seen code that runs smoothly from start to finish. For these you can see, or work out, the exact sequence of instructions that will be carried out once the programme has started to run. However, it is exceedingly common in our numerical methods to control the execution of a programme according to calculations made earlier. In this notebook you will learn how to use to use conditional statements to control the flow of your programme.

In this notebook you will learn how to:

- use boolean statements to control programme execution,
- use the if ... statement to allow alternative code fragments to be executed according to conditions met,
- use if ... elif ... else statements,
- use nested if statements for more complex programme control.

6.1 'if '' statements

6.1.1 Boolean expressions

In python a simple 'if' statement has the form

if *boolean expression* :
indented code

What do we mean by a boolean expression? In boolean algebra a result of a calculation may only have one of two answers: 'True' or 'False' (Boolean alegbra underpins the logic used in all digital computers). So, simply put, if the result of our boolean expression is 'True' the indented code following the _if_ statement is executed, otherwise the statement is ignored and the programme resumes at the next unindented line.

The code below shows some boolean expressions. Look at carefully and run this code taking care to understand the output of each line. In particular note the form of the boolean operators (i.e. _==_ , _!=_ , _>=_ etc)

In [1]:

```
correct=True        # Define a boolean data type with value True
incorrect=False     # Define a boolean data type with value  False
print("Check boolean data types, ",correct, type(correct),incorrect,type(incorrect)) # chec
#
# Now lets define some quantities and look a the results of some boolean operators
#
i=1
j=-1
k=1
a=10.
b=-10.
yes="Y"
no="N"
print ("result of i == j is",(i==j))                 ## == is boolean 'identically equal t
print ("result of i == k is",(i==k))
print ("result of i != j is",(i!=j))                 ## != is boolean 'not equal to'
print ("result of i != k is",(i!=k))
print ("result of i > k is",(i>k))                   ## > is boolean 'greater than'
print ("result of i >= k is",(i>=k))                 ## >= is boolean 'greater than or ide
print ("result of i <= k is",(i<=k))                 ## <= is boolean ' less than or ident
print ( "result of a == b is",(a==b))
print ( "result of a < b is",(a < b))                ## < is boolean 'less than'
print ( "result of a > b is",(a > b))
print ("'yes = N' is",(yes=="N"))                    ## Note string comparisons are case s
print ('"yes = Y" is ',(yes=="Y"))                   ## Note also how we can include quote
```

```
Check boolean data types,  True <class 'bool'> False <class 'bool'>
result of i == j is False
result of i == k is True
result of i != j is True
result of i != k is False
result of i > k is False
result of i >= k is True
result of i <= k is True
result of a == b is False
result of a < b is False
result of a > b is True
'yes = N' is False
"yes = Y" is  True
```

From the above you should now be able to recognise boolean statements. A particular point to note is the difference between for example '<' and '<=' - check you understand the meaning from the results above.

Considerable care shoud be taken when using '==', '>=' and '<='. There is no ambiguity when using integer variables, however, these operators, when used with float variables can easily lead to programming errors. See the code snippet below.

In [2]:

```
from math import *
print(sin(pi))
print(sin(pi)==0)
```

```
1.2246467991473532e-16
False
```

We all know $\sin(\pi) = 0$ so why is the boolean result false? It is rare, in floating point calculations to obtain perfect equivalence between floating point variables. In this case the rounding errors in calculating $\sin(\pi)$ mean the result is not identically equal to zero.

WARNING. Avoid making 'identically equal' equivalence tests with floats. If necessary, convert to integer or use composite boolean statements (see below).

6.1.2 testing and combining boolean expressions.

We can use boolean algebra (truth tables etc) to build composite tests for our vairables. To compare boolean expressions we can use the *and*, _or_, *not* and _^_ operators. If we consider two boolean numbers A and B then we can understand these operators as follows:

The AND operator (*and*) says if both A and B are true then the result is true, else the result is false.

The OR operator (_or_) says if both A or B, or both are true then the result is true, else the result is false.

The XOR operator (_^_) says if A or B are different, the result is true, if they are the same the result is false

The NOT operator (*not*) changes boolean value of a variable to its opposite (i.e. true to false, or false to true.)

Note also the operator precedence for boolean operators, if in doubt use brackets.

Look at the code snippet below and confirm you understand the output. (These are effective truth tables)

In [3]:

```
T=True
F=False
print ("T AND F is",(T and F))
print ("T AND T is",(T and T))
print ("F AND F is",(F and F))
print ("T OR F is",(T or F))
print ("T OR T is",(T or T))
print ("F OR F is",(F or F))
print ("T AND (NOT F) is",(T and not(F)))
print ("T AND (NOT T) is",(T and not(T)))
print ("T OR (NOT F) is",(T or not(F)))
print ("T OR (NOT T) is",(T or not(T)))
print ("T XOR T is",(T ^ T))
print ("T XOR F is",(T ^ F))
print ("F XOR F is",(F ^ F))
```

```
T AND F is False
T AND T is True
F AND F is False
T OR F is True
T OR T is True
F OR F is False
T AND (NOT F) is True
T AND (NOT T) is False
T OR (NOT F) is True
T OR (NOT T) is True
T XOR T is False
T XOR F is True
F XOR F is False
```

6.1.3 the simple 'if' statement

in 6.1.1 we stated that the 'if statement has the form,

if *boolean expression* :
if true execute indented code

Now we know what we mean by a boolean expression we see how the 'if' statement works. Simply put, if I carry out a boolean test, then if the result is True I execute the indented code after the if statement, if it is False I ignore it and resume execution after the indented code.

Look and read carefully the code snippet below and then execute it.

```
In [12]:
```

```python
from math import *
a=2.0
b=3.0
c=-3.0

if a*b >= 0:                #
    print("square root of a*b is ",sqrt(a*b))
if a*c >= 0:
    print(sqrt(a*c))    # as the product a*c is negative this line will not be executed

#print(sqrt(a*c))         # the programme will crash with an error oif we don't do this tes
```

```
square root of a*b is  2.449489742783178
```

This is an example of a common test we may wish to execute in a programme. Here we want to find the square root of the product of two numbers. However, if the product is a negative number our programme will stop with an error message (uncomment the last statement to see what error is produced). In the example above we can see how we can test for a negative number and allow the programme to continue. Of course, we need to decide what we are going to do if we find a negative number.

Look and read carefully the code snippet below,

```
In [13]:
```

```python
from math import *
a = sin(pi)
if a < 1e-10 and a > -1.0e-10 :
    print ("value of a was ",a )
    a=0.0
    print("We have now set the value of a to zero",a)
```

```
value of a was  1.2246467991473532e-16
We have now set the value of a to zero 0.0
```

Note how this solves our problem in floating point arithmetic to check if our variable is zero (or close enough to zero) to be used further in our calculation.

Hence, if you know how to do a boolean test you know how to use the 'if' statement.

6.1.4 the if ... else statement

The _if_ statement is all well and good but suppose we wish to execute different bits of code according to whether our boolean expression is true or false. This is easly solved by the extension of the _if_ statement to include an *else* statement as follows

if *boolean expression* :

indented code

else:

indented code

Look at and run the code fragment below to see how this works.

In [17]:

```
from math import *
a=2.0
b=-3.0
c=a*b
if c > 0:
    c=sqrt(c)
    print("square root of a*b is",c)
else:
    print("I can't find the square root the negative number", c)

print ("End of calculation")
```

```
I can't find the square root the negative number -6.0
End of calculation
```

Hint. It is very easy to forget the colon on both the _if_ and the *else* statement. If you have errors when writing your code it is often a good idea to check for this simple omission.

We can now see that whatever the result of the boolean statement, we can run some code for the case when the statement was true *and* also when it is false. The programme continues after the indented code after the 'else' statement. You might modify the code above to make _b_ positive to see how the execution of the code depends on the sign of _c_. Note, that there is no boolean statement associated with the *else* line.

Note. Some programmers advocate that, for clarity, an else statement should always be included, even it is followed by a null statement.

6.1.5 the if ... elif ... else statement

The _if_ statement can be extended by adding further boolean tests with the *elif:* line. This stands for 'else if' and allows you to test sequentially with a series of boolean tests. The *elif* line must be followed by a boolean statement after which the following indented code is executed if it is true. If it is false the programme moves to the next *elif:* or final *else:* line. It is important to realise that once the programme comes to first test with a true answer, then no further tests in the *if:* - *elsif:* - *else:* block are executed.

Look at carefully run the following code fragment to see how this works

```
In [18]:
```

```python
i=2
if i==0:
    print("i is equal to zero")
elif i==1 :
    print("i is equal to one")
elif i==2 :
    print("i is equal to two")
elif i==3 :
    print("i is equal to three")
elif i < 5 :
    print("i is less than 5")  # note although i is less than 5, this line is not executed
                               # the condition i==2 was met.
else:
    print ("i is greater than three")


print ("End of programme")
```

```
i is equal to two
End of programme
```

Try changing the value of _i_ in the code above and confirm you understand how it works. You can note again that as in the simple _if_ statement it is not necessary for the last statement to be an *else:* line. However, again for clarity is often recommended that it is included.

Look carefully and run the code fragment below.

```
In [8]:
```

```python
i=1
if i==0:
    print("i is equal to zero")
elif i <= 2 :
    print("i is less than or equal to 2")
elif i==1 :
    print("i is equal to one")
elif i==3 :
    print("i is equal to three")
else:
    print ("i is greater than three")


print ("End of programme")
```

```
i is less than or equal to 2
End of programme
```

when you run this code the line print("i is equal to 1") is never executed as the if statement has already found the statement i <= 2 is true so that execution moves onto the next executable unindented line.

6.1.5 nested 'if' statements

As for the case of for loops it is possible to 'nest' if statements.

Look carefully at and run the code fragment below. Try changing _i_ and _j_ to see how it works.

In [9]:

```
i=1
j=5
if i > 0 :
    print("i is greater than 0")
    if j > 0 :
        print ("j is greater than 0")
    elif j <0 :
        print ("j is less than zero")
    else:
        print ("j is equal to 0")
else:
    print ("as i was less than or equal to zero I didn't check j!")

print("programme end")
```

```
i is greater than 0
j is greater than 0
programme end
```

Try different integer values of i and j to see how this works. The same result could have been achieved in a different way. You might like to try a different method. Nevertheless it demonstrates the use of nested if statements.

6.2 Summary

In this notebook you have learnt how to:

- construct and use boolean statements that give a result 'True' or 'False',
- use boolean statements in an _if_ statement to give conditional branching in your programme,
- use the *else* statement to execute code when a true result has not been found in the _if_ statement,
- use the *if: ... elif: ... else:* statement to allow multiple and sequential tests to allow different code to be executed for different cases,
- use nested of statements for more complex tasks.