# Decision structures

***Read:*** *Chapter 7, Sections 8.1-8.2 from Chapter 8 from textbook*

- Decisions about what to do next in a program are based on **logical conditions**: these are conditions that evaluate to either True(T) or False(F). The following are conditions:

```
>>> 3 >= 4
>>> 3 != 4
>>> x = 3
>>> x == 3
```

  The “!=” in the second line means “not equal”. The opposite (equal) is “==”. Note that lines 3 and 4 are completely different animals: the first is an *assignment* statement and the second is a *condition*. In fact, you can think of conditions as another kind of <expression>, only they evaluate to T or F instead of evaluating to a numerical value.
- The simplest decision structure is an if-statement:

```
        if <condition>:
            <body>
```

  The statements in <body> will execute only if <condition> evaluates to T. For example:

```
>>> for i in range(10):
        if (i>5):
            print(“{0} is greater than 5.”.format(i))
        print(“The current i is {0}.”.format(i))
```

- A more complex decision structure is an if-else-statement:

```
        if <condition>:
            <body1>
        else:
            <body2>
```

  If the <condition> evaluates to T then the statements in <body1> execute, else (i.e., if <condition> is F) the statements in <body2> execute. For example:

```
>>> for i in range(10):
        if (i>5):
            print(“{0} is greater than 5.”.format(i))
        else:
            print(“{0} is less or equal to 5.”.format(i))
```

- In general, you can branch according to more than one conditions using elif (else if) as follows:

```
if <condition1>:
    <body1>
elif <condition2>:
    <body2>
elif <condition3>:
    <body3>
elif .....
    ......
else:
    <bodyN>
```

In this structure, the logic is the following: if <condition1> is T, then <body1> executes; else (i.e., <condition1> is F) if <condition2> is T, then <body2> executes, etc. In short, <bodyK> executes only if conditions <condition1>, <condition2>,...,<condition(K-1)> are all F and <conditionK> is T. If all conditions are F then <bodyN> executes (of course, you can omit the else part if in this case you just want to continue with the rest of your program). For example

```
>>> if (i>5):
        print("{0} is greater than 5.".format(i))
    elif (i==5):
        print("{0} is equal to 5.".format(i))
    else:
        print("{0} is less than 5.".format(i))
```

- **Indefinite loops:** Logical (or Boolean) conditions allow us to build more sophisticated loops than the definite for-loops we have already seen. Instead of executing the body of the loop a pre-specified (definite) number of times, we can execute it repeatedly, until a <condition> evaluates to F. This is done with a while-loop:

```
while <condition>:
    <body>
```

Here the <body> is executed again and again until <condition> becomes F (if <condition> is already F the first time, then <body> will not execute at all!). For example:

```
>>> i=0
>>> while i<=10:
        print(i)
        i=i+1
```

Here is a common error that all programmers commit: Suppose that the last line is missing. Then <condition> (here `i<=10`) will always be T, and the loop will execute for ever (an **infinite loop!**) This is one of the main reasons that make an application to freeze; it has fallen in an infinite loop.

- Now that we have covered both definite and indefinite loops, a bit of loop terminology: inside the <body> of a loop you can have another loop, whose <body> also can contain a loop, etc. These loops, one inside another are called **nested loops**.
- **Exception handling:** A special kind of condition that we would like to evaluate and, if T, do something different than what we normally do, is a condition of the form "Did an error occur?" In particular, we would like to be even more specific, and be able to evaluate as T or F something like "Did error X occur?" X here denotes the particular kind of error. In the following examples, type exactly what you see and look at the last line of the message the Python interpreter outputs; it tells you what kind of error happened:

```
>>> from math import *
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

The error you got is a `ValueError`. You got it because you gave a negative number to a function whose domain is all non-negative numbers.

```
 >>> inport math
File "<stdin>", line 1
  inport math
            ^
SyntaxError: invalid syntax
```

The error you got here is a `SyntaxError`, because you typed "inport" instead of "import".

```
>>> [1,2]+3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

The error you got is a `TypeError`, because you tried to apply + to two operands of incompatible types (the perennial apples and oranges problem).

```
>>> GKarakostas
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'GKarakostas' is not defined
```

Here the error you get is a `NameError`, not because the name is wrong (it is correct), but because it is a name you have not defined (and, therefore, you can not enquire about its value).

Python allows you to put a piece of code under constant lookout for errors (which is a form of

**exception** from normalcy); in case the particular error(s) occur, then you can specify what to do for each kind of error you care to **handle.** The general structure is the following:

```
try:
    <body>
except <error_type>:
    <handler>
```

The piece of code under supervision is <body>; if an error of type <error_type> (e.g., TypeError), then the execution of <body> stops and <handler> is executed instead. As an example, look at program quadratic5.py (p. 216 in the text). In this example, the <body> is calculating the real solutions to a quadratic equation, and the error we are at the lookout for is ValueError; if it happens, then the handler is just the last print statement. The program quadratic6.py (p. 218) captures more types of error.

**Practice problems**

1. Do all the problems in the textbook.
2. Do the Discussion question 2 in p. 229 (not by typing the code, but by tracing it yourself by hand; this is supposed to train you mentally on the logic of decision structures and conditions).
3. Write a program that asks the user for a number x. Then it outputs the following menu, followed by a request for a choice between 1-4:

    1. Square
    2. Square root
    3. Sin
    4. Factorial

    Please enter a choice 1-4:

    After the user enter a choice, call the corresponding mathematical function (some are included in the math library, some are implemented by you). The program must check whether the choice provided by the user is a number between 1-4; if it isn't, then it should output a message before quitting.
4. Change the previous program by adding exception handling in order to capture errors due to corrupt input by the user; in other words, use the try/except structure like it is done in quadratic6.py.
5. Change the previous program to add a new menu option:
    0. Quit

    The program should run continuously until the user chooses option 0 (or, of course, an error occurs).
6. (This problem is intended as an invitation for thought; we will study recursion thoroughly later.) Recall the recursive definition of factorial from a previous lab:

```
fact(1) = 1
fact(n) = fact(n-1) * n,  if n>1
```

This definition is recursive because the function fact() is defined using itself (!) in case n>1 (the case n=1, called the base case, is defined in a more normal way...) So, you should be wondering what kind of definition is this, where something is defined in terms of itself. Isn't this cyclical? For now, implement the function fact() not by using a for-loop (as we did before), but by mimicking the definition (with call to itself and all...) Notice that it works! Now, add the original for-loop implementation in your module (call it, say, fact-l()). Run both versions of factorial for a number provided by the user in your program; first call the loop version and then the recursive version. The two results produced must be the same (obviously!). Now, try to give as an input bigger and bigger numbers; what do you notice about the running time of the two versions?

7. Write a (rather silly) program that asks the user for a float x, and then checks whether
$$(\sqrt{x})^2 = x$$
If the LHS (Left-Hand Side) is equal to the RHS, then it should output "All right!", otherwise it should output "Bummer!" Now run your program; if your condition is a simple equality condition of the form 'LHS==RHS', most likely you got a "Bummer!". Why?
8. Does the previous problem imply that we will never be able to reliably compare two floats for equality? *(Hint: Maybe we should change our test; instead of checking for absolute equality, maybe we should check for "close enough". This means that we will be content if the difference LHS-RHS is a very small (positive or negative) number, say, between -p and p for a very small p. Try to change your test to follow this new condition for, say, p=0.0000000001. You may find the function abs() useful...)*