## Contents

## 1 The Python interpreter

- The first step to learning programming in Python is to have a working Python interpreter.
- The course will be based on Python 3, i.e. we will assume that you are running your programs on an interpreter with a version number starting with 3.[1]
- Please visit https://docs.python.org/2/using/index.html for installing Python on the platform of your choice.
- You can interact with the interpreter, you can have it run a stored program, or you can do both.
- The previous item may not mean much at the moment, don't worry.
- When you invoke the Python interpreter on its own, you will enter into the **interactive mode** of the interpreter.

```
Python 3.4.2 (default, Oct  8 2014, 13:14:40)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- You can see the version you are running by checking the number in the first line, following '`Python`'.
- The Python **prompt** '>>>' indicates that the interpreter is ready to receive input from the **user**.
- From now on, I will call the interface by which you interact with the interpreter the **Python console**, or simply the **console**.
- There are two ways to **exit** the console. You can either type `exit()` followed by hitting `enter`, or you can hold down the control key and press `D`.

[1] Please check https://docs.python.org/3.0/whatsnew/3.0.html for what is new in Python 3.

## 2 Arithmetic

- A fundamental distinction you need to learn well is the one between **expressions** and **statements**. Here we start with expressions. The statements will come later below.
- An expression is a piece of code that can be read by the interpreter. The interpreter deciphers this code – or understands it, if you like – and computes the **value** of this expression, and finally returns the value.
- To fix the terminology, interpreter evaluates an expression and computes a result. This computed result is the value of the expression.
- You can type an expression on the Python console and hit `enter` for the interpreter to **evaluate** the expression and **print** the result on the screen. The simplest kind of expression you can provide to the interpreter is a number. Simply type it on the console and hit `return`.

```
>>> 4
4
```

- There is nothing more basic than an integer, therefore the interpreter give you the integer back as the result of its evaluation.
- Arithmetic expressions are so fundamental to computing that almost all programming languages have **constructs** to perform arithmetic. You can actually use the Python interpreter as a calculator; type an arithmetic expression to the interpreter and hit `enter` to let the interpreter evaluate the expression and return the result:

```
>>> 4 + 7
11
```

- Now try something more complicated:

```
>>> 4 + 7 * 8
60
```

- This could be a little surprise for you, if you were expecting the result to be 88 rather than 60. Actually 88 would be the result for an interpreter that evaluates arithmetic expressions from left to right, without caring about the kind of the arithmetic operator it is interpreting. The Python interpreter, like other interpreters, has some grouping rules concerning algebraic expressions like arithmetic operations. For instance, we have just discovered that it gives **precedence** to multiplication with respect to addition. If you want to compute first the sum of 4 and 7 and then multiply the result by 8, you should type:

```
>>> (4 + 7) * 8
88
```

- Some common operations that are available in the Python console are:[2]

| Operator | Name | Example |
|---|---|---|
| `+`,`-`, `*` | add., subt., mult. | |
| `/` | division | '`8/5`' gives '`1.6`' |
| `//` | floor division | '`8//5`' gives '`1`', round the result to the greatest lower integer |
| `%` | modulo | '`8%3`' gives '`2`', the remainder of division |
| `**` | exponentiation | '`8**3`' gives '`512`' |

**Exercise 2.1.**
Open up a Python console, and by trying various expressions, figure out the grouping rules regulating the precedence of the operators given in the above table.

## 3 Data types

- With some idealization, you can think of computation as some actions operating on data and generating new data. The data can be integers, floating point numbers (floats), characters, strings, lists, matrices, and so on.[3]
- All the types of data mentioned above have their specific instances. As we will see shortly, the expression `4` is an instance of the type integer. We will call these instances **objects**. We will have more to say about these matters later, but let us fix the terminology from the beginning: Every **object** is an instance of a **class**, which names the object's **type**. If you are confused about objects, classes, and so on, at this point, do not worry and hold on.
- Every programming language comes with a collection of **built-in data types**. What we mean by being built-in is that the language has some basic expressions to represent the type of data in question.[4]

[2]Note that in Python 2, '`/`' performs floor division, if the operands are integer; more on types below.
[3]We will not cover the numerical type `complex`.
[4]Some analogy: In Turkish you have a built-in expression to ask the place of an item in an ordered sequence, namely *kaçıncı*. English lacks such a built-in expression, it needs to construct complex expressions to ask the same question.

## 3.1 Integer

- So far we have seen only one of the built-in data types of Python, namely integers.
- You can ask the type of the value of an expression – which is an object – on the Python console, by using a **built-in function** called ‘type’.

```
>>> type(4)
<class 'int'>
```

- What type does is to first let the interpreter evaluate the expression in the parentheses, and then checks the type of the result of this evaluation.

```
>>> type(4*2**3)
<class 'int'>
```

## 3.2 Float

- Another numerical type is that of floating point numbers, or simply **float**.

```
>>> type(4.4)
<class 'float'>
```

```
>>> type(8/5)
<class 'float'>
```

- Here are some more built-in functions that operate on numbers.

```
>>> abs(-4.2)
4.2
>>> pow(3, 2)
9
>>> divmod(8, 3)
(2, 2)
```

- If what divmod does is not clear, try to figure it out by giving it other arguments.
- A handy function to be used with float objects is round:

```
>>> round(44.6)
45
>>> round(44.5)
44
>>> round(3.141592653589793,4)
3.1416
```

- Here we observe that there are two functions with the name round. One rounds the floating number to the nearest integer, and takes only the floating number to be rounded as argument. The other round takes two arguments. The first is the float, the second is the number of digits to be kept after the floating point.

## 3.3 String

- Deep down, computers operate via numerical representations. However, computers are used in many tasks that are not numerical – at least from the perspective of the user, take a text editor, web-browser, or a library management system for instance; programming itself is performed by mainly words.
- A string is an ordered sequence of *zero or more* characters.[5] The type of such objects is **str** in Python. There are two ways to represent strings: either with matching single quotes ‘''’, or by matching double quotes ‘"’. All the following are strings:

```
>>> type('abracadabra')
<class 'str'>
>>> type("a")
<class 'str'>
>>> type(' ')
<class 'str'>
>>> type('')
<class 'str'>
```

- Just like numbers, strings can be added by ‘+’ and you can “multiply” a string by an integer:

```
>>> 'abra' + 'cad' + 'abra'
'abracadabra'
>>> 'cad' * 3
'cadcadcad'
```

[5] In Python, there is no basic data type for characters; they are treated simply as unary strings.

```
>>> 3 * 'cad'
'cadcadcad'
```

- As we observed above, the built-in function `type` is evoked by providing its argument within parentheses. Python has another kind of built-in function which is called a **method**. These functions operate relative to certain types of objects. For instance a string object supports the method `upper()`, which, like other methods, is evoked by the following syntax:

```
>>> "abracadabra".upper()
'ABRACADABRA'
```

- We will encounter more string methods below.

## 3.4 Boolean

- We have seen above that arithmetical expressions evaluate to number objects and string concatenation or multiplication evaluate to string objects. Some expressions are like yes-no questions, they evaluate to true or false. Python has two designated objects that stand for these truth-values: `True` and `False`. Their type is **bool** (shortened from 'boolean').

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

- Yes-no questions are asked by **comparison operators**. All but the two[6] are:

| | |
|---|---|
| < | strictly less than |
| <= | less than or equal |
| > | strictly greater than |
| >= | greater than or equal |
| == | equal |
| != | not equal |

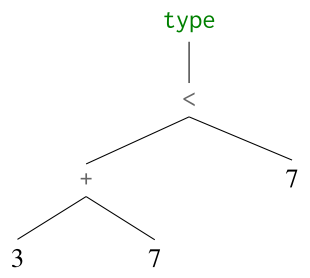
- Comparison operators can be used to compare both numbers and strings. For strings alphabetical order determines the result of the comparison. Try

```
>>> 'abra' < 'cad'
True
>>> 8 != 7
True
>>> 8 == 7
False
```

- Now let us combine some of the concepts we have encountered so far:

```
>>> type(3 + 7 < 7)
<class 'bool'>
```

- Here, as in arithmetic, Python interpreter observes the precedence relation among operators. It first **evaluates** the addition operator since comparison operators has lower precedence than arithmetic operators; then the comparison is performed – giving `False`, and finally the built-in type detecting function is applied to `False`, giving the type expression for boolean, namely `<class 'bool'>`. The evaluation order can be pictured as a tree.



- Boolean values can be combined with boolean operators **`and`**, **`or`**, and **`not`** as in symbolic logic (more on this below).

## 3.5 Type conversion (manual)

- You can convert between types using **type constructors**, which are built-in Python functions. The constructors for the data types we have seen so far are `str`, `int`, `float` and `bool`. They take their arguments in parentheses. Some sample conversions are:

```
>>> str(3.4)
'3.4'
>>> float(3)
```

[6] We will discuss object comparison operators later.

```
3.0
>>> int(3.7)
3
```

## 4 Variables, identifiers, assignment statements

- A **variable** is a location where data – Python objects in our case – is stored. The content of a variable may change, hence the name "variable".
- It is useful to have names for things, and variables are no exception. We can name (or point to) variables by **identifiers**. Python allows as identifier any sequence of letters, numbers and the underscore character ('_'), provided that the first character in the sequence is not a number or the sequence is not a special expression reserved for a designated use in Python, like `print` for instance; such expressions are called **keywords**.
- By using the identifier you can have access to the information (what we have been calling object or value) stored in the variable. We simply say that an identifier points to an object, or refers to an object, skipping the detail about the variable that mediates this relation. You can make an identifier point to an object by an **assignment statement**:

```
>>> my_number = 5
```

- From there on, you can use `my_number` everywhere you want to use 5.
- It is very important to learn the meaning of '=' in programming, as it is quite different from what we normally understand from equality in mathematics.[7]
- Let's assume you continue the above code with entering this:

```
>>> your_number = my_number
```

- Now the following tests should succeed:

```
>>> your_number == 5
True
>>> your_number == my_number
True
>>> my_number == your_number
True
```

- Now modify `my_number`:

```
>>> my_number = 7
```

- What do you expect as the value of `your_number` when you modified the value of `my_number`? 5 or 7? Think a little before you try it on the console. (Get the habit of thinking before trying, this is very important in learning programming.)
- You will discover that modifying `my_number` has no effect on `your_number`. This is because what is designated by '=' is not equality but the relation of "pointing to". When you say:[8]

```
<identifier> = <value>
```

  you mean that the name `<identifier>` points to the object `value`. After that whenever you use the name `<identifier>` in an expression, it gets evaluated to the object/value it points to. On the other hand, when you say:

```
<identifier1> = <identifier2>
```

  you mean that the name `<identifier1>` points to the same object/value that the name `<identifier2>` points to. You do not establish a direct connection between the two identifiers; therefore manipulating one, does not effect the other.[9]

- Another important point about assignment statements is the order in which the Python interpreter handles them. First the expression on the right hand side is evaluated, *then* the identifier on the left hand side is made pointing to that value. One nice corollary of this is the possibility of statements like the following:

[7] Some programming languages use '<-' instead.

[8] Here is a useful convention: Enclosing names between angle brackets forms a meta variable over the type of thing designated by the name. E.g. "`<identifier>`" means "take an arbitrary identifier", likewise "`value`" stands for an arbitrary value.

[9] Take the sentence 'John goes shopping on the same day as Mary'. The two possible readings of this sentence are:

(R1) Mary goes to shopping on a certain day (say Monday), and that day is also the day for John to go shopping.

(R2) Whichever day Mary goes shopping, John does too on that day; John follows Mary's shopping schedule day-to-day.

The relation between identifiers we are concerned here is of type R1 rather than R2. This relates to the issue of extension versus intension.

```
>>> my_number = 8*(5-3)
>>> my_number
16
>>> my_number = my_number + 4
>>> my_number
20
```

- It works exactly the same with other operations and other data types.

```
>>> my_word = 'abra'
>>> my_word
'abra'
>>> my_word = my_word * 4
>>> my_word
'abraabraabraabra'
```

- We have seen some assignment statements; pieces of code that put the '=' sign in between other things. It is important to notice how such statements differ from expressions. Take the following code:

```
>>> my_number = 8*(5-3)
```

  when you enter this into console you get an empty prompt; in other words the console does not return anything. There is still an expression, `8*(5-3)`, as part of the statement and it gets evaluated; but the assignment statement itself does not return any value; it simply makes `my_number` point to the value computed by evaluating `8*(5-3)`.

- In general, a statement is a piece of code that has a side effect; it does something beyond evaluation of expressions. Statements are executed, expressions are evaluated. You can think of a statement as an imperative sentence. The distinction will get clarified as we go along, just keep an eye on the distinction.

## 5 Simple input/output

- Some computer programs interact with their users. A basic interaction is to ask for an input from the user and to display an output for the user. The simplest way is using an `input()` and `print()` statement, respectively:

```
>>> my_number = input()
'8'
>>> print(my_number)
'8'
>>>
```

- You could use `input()` on its own as a statement – try it. In that case it still asks for an input to be entered, but the entered input is inaccessible afterwards. What we did above is to store the value entered by the user in a variable named `my_number`. This enabled us to use the value for some other purpose later on.

- You can make `input()` and `print()` more "communicative" by providing them string **arguments**:

```
>>> my_number = input('Give me a number: ')
Give me a number: 8
>>> print('My favorite number is ',my_number)
My favorite number is  8
>>>
```

- As you might have noticed, `input` returns a string even the user has entered an integer. Python3's `input` takes every input as a string. If you want `input` to read what is provided by the user as if the input is entered to the console, you **wrap** the function `eval` around `input`:

```
>>> my_number = eval(input())
8
>>> print(my_number)
8
>>> type(my_number)
<class 'int'>
>>>
```

## 6 Programs

- In languages like Python, a program is a sequence of one or more statements.[10] Although it is possible to enter your program line-by-line to the console, it is much more convenient to have it stored in a file and then run by the Python interpreter. Store your python programs in text files with the extension `.py`.

  **Exercise 6.1.**

  1. Write a program that asks for your name and prints the number of letters in it.
  2. Write a program that asks for your surname and prints its 2nd and 5th letters. (We maintain that no surname has less than 5 letters by filling the empty slots with 'x'.)
  3. Write a program that asks for two integers and prints their product.
  4. Write a program that asks for a word, and prints a version of it where the first letter is changed to 'C'.
  5. Write a program that asks for a word, and prints a version of it where the first and the last letter is in upper-case.
  6. Write a program that asks for a string and prints 0 if it has an even length and 1 if it has an odd length. Do not use `if`.
  7. Write a program that asks for a string and prints `True` if its 2nd and 4th symbols are equal. Do not use `if`.

## 7 Containers

### 7.1 Lists

- A Python **list** is a comma separated sequence of objects enclosed in square brackets.

```
>>> a = [1,2,3]
>>> type(a)
<class 'list'>
>>> b = list('abcd')
>>> b
['a', 'b', 'c', 'd']
```

- String indexing works for lists as well – make sure you fully understand how all the expressions below are evaluated.

[10]Some people like to use the word "script" instead of "program" for Python programs. The distinction is not important for us.

```
>>> things =  [2,3,'x',7,'yz',13]
>>> things[5]
13
>>> things[2:5:2]
['x', 'yz']
>>> things[2:5:2][1]
'yz'
>>> things[2:5:2][1][1]
'z'
```

- In contrast to strings, lists are mutable; you can change any element of a set by using the corresponding index; but you cannot add an element to a list in this way – try.

```
>>> things =  [2,3,'x',7,'yz',13]
>>> things[5]= ['u',8]
>>> things
[2, 3, 'x', 7, 'yz', ['u', 8]]
```

- Note that a list can contain any type of object, including lists.
- The concatenation operator '+' used for stings is applicable also to lists.

```
>>> things =  [2,3,'x',7,'yz',13]
>>> more_things = ['u',8]
>>> things + more_things
[2, 3, 'x', 7, 'yz', 13, 'u', 8]
```

- The concatenation operator '+' returns the result of concatenation of its operands, without changing the operands themselves. After the concatenation above, the lists `things` and `more_things` stay the same as before. If you want to change a list by concatenating another list to it, you have two options:

  Option 1:

```
>>> things =  [2,3,'x',7,'yz',13]
>>> more_things = ['u',8]
>>> things = things + more_things
>>> things
[2, 3, 'x', 7, 'yz', 13, 'u', 8]
```

Option 2:

```
>>> things =  [2,3,'x',7,'yz',13]
>>> more_things = ['u',8]
>>> things.extend(more_things)
>>> things
[2, 3, 'x', 7, 'yz', 13, 'u', 8]
```

- The extend() method is easy to confuse with append(). The latter adds its argument to the list it runs on:

```
>>> w =  [2,3,7]
>>> w.append('x')
>>> w
[2, 3, 7, 'x']
>>> w.append(['x'])
>>> w
[2, 3, 7, 'x', ['x']]
>>> w.extend(['x'])
>>> w
[2, 3, 7, 'x', ['x'], 'x']
```

therefore extending a list *L* with some list $[a]$ is equivalent to appending *a* to *L*.

- Lists allow for multiple assignments:

```
>>> w = [8,0]
>>> x, y = w
>>> print(x,y)
8 0
```

- The test for membership is done as follows:

```
>>> w = [8,0]
>>> 7 in w
False
>>> 8 in w
True
>>> v = 3
>>> v in [1,2,3]
True
```

- A useful method for string to list conversion is split():

```
>>> s = '1;2;3;4'
>>> s.split(';')
['1', '2', '3', '4']
>>> t= '1 2 3 4'
>>> t.split(' ')
['1', '2', '3', '4']
```

- The dual of split() is join():

```
>>> t= ['a','b','c','d']
>>> '-'.join(t)
'a-b-c-d'
```

Note that join works only with lists of strings.

## 7.2 Tuples

- A Python **tuple** is an immutable list.

```
>>> w = 1,4,2
>>> type(w)
<class 'tuple'>
>>> w
(1, 4, 2)
>>> u = tuple([1,4,2])
>>> u
(1, 4, 2)
>>> v = tuple('abcde')
>>> v
('a', 'b', 'c', 'd', 'e')
```

- Except methods and operations that are aimed to change the object, whatever you can do with a list applies for a tuple as well.

- Advice: Use tuples instead of lists, (only) when there is a risk of accidentally altering the contents and/or size of the container.

### 7.3 Dictionaries

- A Python **dict** is a set of key-value pairs:

```
>>> employer32 = {'Name':'J. Smith', 'Department':'Pretty Things'}
>>> employer32['Department']
'Pretty Things'
```

### 7.4 Sets

- Python has a **set** data type that implements the mathematical concept of sets. Use `help(set)` to see how this data type works.

- Use sets instead of lists, if repetitions and order is not important for the task at hand.

## 8 Flow of control

- We will cover 3 basic ways to control the flow of execution:

  1. sequencing;
  2. conditional branching;
  3. looping, which takes the two forms:
     (a) bounded looping;
     (b) conditional looping.

### 8.1 Sequencing

- This is the most basic control structure, and it is also simple to state: in the absence of other control structures statements are executed in the order they are encountered going from top to bottom.

### 8.2 Conditional branching

- A simple conditional statement is of the form

  **if** <*boolean expression*> **then** <*statements*>

  where <*statements*> are executed if <*boolean expression*> gets evaluated as `True`, and skipped otherwise. The syntax of the expression requires some care:

```
n = int(input('An integer please? '))

if n%2 == 0:
    print('You entered an even integer')
```

- The above code tells the interpreter what to do when the provided boolean test returns `True`; you can specify what to do for the `False` case as well.

```
n = int(input('An integer please? '))

if n%2 == 0:
    print('You entered an even integer')
else:
    print('You entered an odd integer')
```

### 8.3 Looping

- Looping allows for repeating an action with different parameters at every repetition.

#### 8.3.1 Bounded loops

- The maximum number of repetitions is fixed. Python uses the `for` structure for this.

```
a_list = input('A space-separated list: ').split(' ')

for x in a_list:
    print(x)
```

#### 8.3.2 Conditional loops

- The second type of looping does not have an upper bound for the number of repetitions. The loop runs as long as a condition is met. Python has the `while` construct for this purpose:

```
n = input('An integer please? ("q" to quit) ')

while n != 'q':
    n = int(n)

    if n%2 == 0:
        print('You entered an even integer')
    else:
        print('You entered an odd integer')

    n = input('An integer please? ("q" to quit) ')

print('Bye!')
```

## 9 Functions

- A Python function is a subprogram that can be called within other programs, including itself.
- For instance, assume you have the task of deciding whether two lists of numbers have the same arithmetic mean. Here is a first attempt:

```
list1 = eval(input('A list of ints please: '))
list2 = eval(input('Another list of ints please: '))

sum1 = 0
for i in list1:
    sum1 = sum1 + i

mean1 = sum1/len(list1)

sum2 = 0
for i in list2:
    sum2 = sum2 + i

mean2 = sum2/len(list2)

if mean1 == mean2:
    print('Means match!')
else:
    print('Means do not match!')
```

- The above program can be improved by putting the lists themselves in a for loop, thereby writing the code for mean calculation only once.

```
list1 = eval(input('A list of ints please: '))
list2 = eval(input('Another list of ints please: '))

means =[]

for k in [list1,list2]:
    sum = 0
    for i in k:
        sum = sum + i

    mean = sum/len(k)
    means.append(mean)

if means[0] == means[1]:
    print('Means match!')
else:
    print('Means do not match!')
```

- We will now separate the calculation of the mean from the main program by defining a function that computes the mean of a given list of numbers. First, the function definition in the Python way:

```
def mean(k):
    sum = 0
    for i in k:
        sum = sum +i
    return sum/len(k)
```

- Now the complete program:

```
def mean(k):
    sum = 0
    for i in k:
        sum = sum +i
    return sum/len(k)

list1 = eval(input('A list of ints please: '))
list2 = eval(input('Another list of ints please: '))

if mean(list1) == mean(list2):
    print('Means match!')
else:
    print('Means do not match!')
```

- Organizing your programs into functions is good because:
  - it makes the program easy to write/understand;
  - the function can be re-used within the same program or by other programs.
- Python functions come in three types:
  - functions that return a value, as above;
  - functions that has a side effect without returning anything;
  - functions that do both.
- A function **terminates** (or **halts**) when:
  1. a `return` statement is executed;
  2. an exception that sends the control to outside of the function occurs;[11]
  3. the last statement in the function definition is executed.
- Names internal to a function are invisible to and irrelevant for the users of that function. The case is exactly like in mathematics. Take the function that sums the squares of two numbers:

  $$f(x,y) = x^2 + y^2$$

  The function may well have been defined as:

  $$f(s,t) = s^2 + t^2$$

  Which way it is defined is totally irrelevant for its use.
- This brings us to the notion of **scope** of a variable. Take the following simple functions:

```
def g(y,x):
    return y - x

def f(x,y):
    return g(x,y) / (x + y)
```

[11]This will get clarified when we cover errors and exceptions.

What would be the result of `f(4,6)`?

When the function `f` is called with these **parameters**, the following assignments are made:

$$\mathtt{x} \leftarrow 4$$
$$\mathtt{y} \leftarrow 6$$

When `g` is called with the parameters `x` and `y` from within `f`, what happens is that the identifier `y` in the definition of `g` is made to point to the value that was being pointed to by the identifier `x` of `f`.[12] Pretty confusing. In order not to get confused, in referring to an identifier, let us indicate the owner function as a subscript. When `f(4,6)` is invoked we have:

$$\mathtt{x}_{\mathtt{f}} \leftarrow 4$$
$$\mathtt{y}_{\mathtt{f}} \leftarrow 6$$

- As `g(x,y)` is invoked, the variable assignments become like this:

$$\mathtt{x}_{\mathtt{f}} \leftarrow 4$$
$$\mathtt{y}_{\mathtt{f}} \leftarrow 6$$
$$\mathtt{x}_{\mathtt{g}} \leftarrow 6$$
$$\mathtt{y}_{\mathtt{g}} \leftarrow 4$$

- This shows that the naming of the variables in `g` is totally independent of those of `f`. In invoking a function what is important is the position of the parameters. The first identifier in the function definition is made to point to the value of the first parameter in the function call, the second identifier points to the value of the second parameter, and so on.
- A functional parameter or any identifier that is created within the body of a function does not exist outside the body of that function.
- You can think of each identifier as having an owner. This could be the main program, a function within the program, or a construct like a **for** loop. The rule to keep in mind is that an identifier always belongs to the closest owner.

[12]Strictly speaking, these identifiers are created afresh every time the function is invoked.

## 10 Some basic programming techniques

### 10.1 Counters

- Counting is perhaps the first thing that comes to mind about computing. Keeping a variable that is incremented or decremented according to certain conditions checked within a loop is a ubiquitous programming technique.

- A very simple example would be a function that takes a list and an object, returning the number of times the object occurs in the list.

```python
def occurs_count(lst,obj):
    count = 0
    for x in lst:
        if x == obj:
            count += 1
    return count
```

### 10.2 Accumulators

- Assume you do not have multiplication as a primitive operator, i.e. no `*`. Given two non-negative integers *m* and *n*, you can think of their multiplication as adding *m* to 0 for *n* times. Here is a program that **implements** this idea.

```python
m = int(input('A non-negative integer please: '))
n = int(input('Another non-negative integer please: '))

product = 0

while n > 0:
    product += m
    n -= 1

print('Product: ' + str(product))
```

- Try multiplying 12 with 6666666666 (twelve 6's in a row), first directly on the Python interpreter, then by the program above.

### 10.3 Accumulators

### 10.4 Windows

- Compute the Fibonacci numbers less than 100.

### 10.5 Flags

## 11 Recursion

- A very effective programming technique is **recursion**. A recursive function includes a call to itself. The paradigm example is computing the factorial of a number. Let us first examine an **iterative** algorithm for the factorial function:

```
function FACTORIAL($n$)
    $result \leftarrow 1$
    $count \leftarrow 0$
    while $count < n$ do
        $count \leftarrow count + 1$
        $result \leftarrow result \times count$
    return $result$
```

- Now a recursive algorithm for computing the same function.

```
function FACTORIAL($n$)
    if $n = 0$ then
        return 1
    else
        return $n *$ FACTORIAL($n - 1$)
```

- Note that the recursive algorithm is easier to understand.

**Exercise 11.1.**

a. Write Python code that implements the iterative algorithm for the factorial; write two versions, one with `while` and the other with `for`.

**Solution:**

```python
def factorial(n):
    result = 1
    count = 0
    while count < n:
        count += 1
        result = result * count
    return result
```

```python
def factorial(n):
    result = 1
    for count in range(n):
        result = result * (count + 1)
    return result
```

b. Write Python code implementing the recursive algorithm.

**Solution:**

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

- A more advanced example of recursion is the solution to the ancient game **Towers of Hanoi**.

- The game involves a number of disks with holes in the middle, by which they can be fitted into pegs (or sticks). There are three pegs in total, call them *A*, *B* and *C*. Initially all the disks are fitted to one of the disks in an order of decreasing size – smallest on top. You can move one disk at a time from peg to peg. In no state of the game a larger disk can sit on top of a smaller one. The aim of the game is to transfer the stack of disks from its initial peg to another peg.

- Two questions that immediately arise about the game: Given *n* disks:

  i. What is the minimum number of moves to transfer the disks to another peg;

  ii. What are the specific moves for the same task.

- Work out the problem by hand for $n = 3$ and $n = 4$. This might reveal a recursive solution to the problem. Give yourself some time to work on the solution before reading on.

- Once you work out the solution for $n = 3$, the solution for $n = 4$ is simply running the solution for $n = 3$ once, moving the 3 disks to another location; then moving the 4th disk to the location which is empty, and finally running the solution for $n = 3$ once more to bring the 3 disks on top of the 4th. Therefore, the number of moves you have in total is two times the number of moves for moving 3 disks plus 1. If this works for 4 disks, why should not it work for 5 disks: first move the top 4 to another location, then the 5th, then the 4 on top of the 5th. You can also use the same method to compute the result for 3 disks: 2 times the number of moves for 2 disks plus 1. It is not hard to see that the method works for an arbitrary integer *n* greater than 1, no matter how large it is. The only case you need to dictate the result rather than use the method is $n = 1$, where the number of moves is simply 1. Here is an algorithm to compute the number of moves for any *n*:

```
function NUMBER-OF-MOVES($n$)    ▷ compute number of moves for $n$ disks
    if $n = 1$ then
        return 1
    else
        return (2× NUMBER-OF-MOVES($n - 1$)) +1
```

- Constructing a function that gives you the specific moves is a harder task. As a hint to the solution, think of a function with 4 parameters: move *n* disks from *A* to *B* by using *C*. Try for some time, before reading on.

```
function MOVE($n,X,Y,Z$)                      ▷ move $n$ disks from $X$ to $Y$ using $Z$
    if $n = 1$ then
        print 'Move from $X$ to $Y$'
    else
        MOVE($n - 1,X,Z,Y$)                   ▷ move $n - 1$ disks from $X$ to $Z$ using $Y$
        print 'Move from $X$ to $Y$'
        MOVE($n - 1,Z,Y,X$)                   ▷ move $n - 1$ disks from $Z$ to $Y$ using $X$
```

**Exercise 11.2.**

a. Implement the recursive algorithm for computing the number of moves for Towers of Hanoi in Python. How does your counting program behave as the number of disks gets larger?

b. Implement the recursive algorithm for computing the moves for Towers of Hanoi in Python. A sample interaction with such a program would look like:

```
[code]$ python -i hanoi_moves_rec.py
>>> moves(3,'A','B','C')
Move A to B
Move A to C
Move B to C
Move A to B
Move C to A
Move C to B
Move A to B
>>>
```

**Solution:**

```python
def n_of_moves(n):
    if n == 1:
        return 1
    else:
        return 2 * n_of_moves(n-1) + 1
```

```python
def moves(n,x,y,z):
    if n == 1:
        print('Move '+str(x)+' to '+str(y))
    else:
        moves(n-1,x,z,y)
        print('Move '+str(x)+' to '+str(y))
        moves(n-1,z,y,x)
```

- As you might have realized thinking of the solution and implementing it is easier than actually understanding how it works.
- In order to better understand how the algorithm works, you can either trace the computation by pencil and paper, or use a visual debugger. TODO: Debugger section.