Contents

1 The Python interpreter

- The first step into 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 **expressions** and **statements**. Here we start with expressions. The statements will come later.
- 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 (a string representation of) 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
```

- Interpreter sees nothing to further evaluate given an integer, therefore it gives you the integer back as the result of the 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**. This much will do for the moment.

- 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.[5]

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.[6] 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 '+', the **concatenation** operator, and you can "multiply" a string by an integer:

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

- You can check all the methods available for strings in Python by typing `help(str)` in the console. You can leave the help screen by pressing `q`.

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[7] are:

`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal

[5] Actually it is slightly more complicated than this. Strictly speaking there is one `round` and it has two arguments. What happens when you call `round` with a single argument is that the second argument is looked up from the function specification, which is set to 0 by default. More on this in Section 9.

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

[7] We will discuss object comparison operators later.

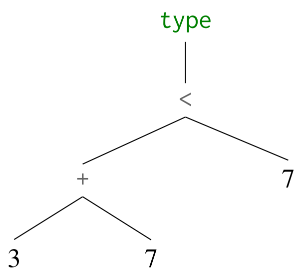
- Comparison operators can be used to compare both numbers and strings. For strings alphabetical order determines the result of the comparison. Try

```
>>> '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)
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**:

```
>>> a_word = "I am not a word, I am a sentence."
```

- From there on, you can use `a_word` everywhere you want to use the string assigned to it. You can go on entering code into console as:

```
>>> len(a_word)
33
>>> a_word.isalpha()
False
>>> a_word.startswith("I am")
True
```

- We encounter some new concepts here. First, you can get the length of a string by the built-in function `len`. Second the method `isalpha`[8] returns a `Bool` type value –

[8] You can check what it does by `help(str.isalpha)`.

the method applies to some type and returns a value of another type. Third, methods, being functions, can take arguments: `startswith` takes a string as an argument and checks whether that string is a prefix of the string that the method is run on.

- 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.[9]

- Let's assume you made this:

```
>>> my_number = 5
>>> 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:[10]

```
<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.[11]

- 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:

```
>>> 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'
```

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

[10] 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.

[11] 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.

- 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 statements (and as we see later function and class definitions).[12] 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`, and run it by either entering `python3 <your program>.py` replacing `<your program>` with the actual name of your program file, or click the appropriate menu item, if you are using a graphical environment.

  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' when entering the input string.)
  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'. (Type `help(str)` in the console to see some useful string methods.)
  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.

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

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`, use `%`.
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.

```
>>> 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 list 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:
- The program in Code 8.1 illustrates a typical use of a **`while`** loop.

Example 8.1 (While loop.)

```
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!')
```

Exercise 8.2

1. Write a program (`flip`) that takes a string as input and gives another string which has the same characters as the input except the case of every cased character (i.e. letters) is altered. It would turn `'abRaCadABra'` to `'ABrAcADabRA'`, `'Aa+aA'` to `'aA+Aa'`, and so on.
   You may traverse a string letter by letter by using a `for` loop, exactly like how we do for lists. Each iteration gives a one character string from the original string in the order of their appearance.
   You may use the built-in string functions `isupper()`, `islower()`, `upper()`, `lower()` in your program.
2. Write a program (`acro`) that takes a string and returns an acronym. For instance it should return `'LSD'` for `'Lucy in the Sky with Diamonds'`.
   But be careful, it should NOT return `'ISIS'` for
   `'mIddle eaSt technIcal univerSity'`,
   but print `'Sorry, no acronym for that.'` instead. You may use the built-in functions mentioned in the previous question.

3. Write a program (`zipp`) that takes two lists and returns a list formed by taking one element from the first, then one from the second, then from the first, and so on. The program should be able to handle cases where one list is shorter than the other. Example outputs would be `[1, 5, 2, 6, 3, 7, 4, 8]` for input lists `[1,2,3,4]` and `[5,6,7,8]`, or for example `[1, 5, 2, 6, 7, 8]` for input lists `[1,2]` and `[5,6,7,8]`.
4. Write a program (`flat`) that takes a possibly nested list (= a list with at least one element which is itself a list) and returns a non-nested list. It gives `[1,2,3,4,5]` for `[[1,2],3,[4],5]`, or `[1,2,3]` for `[[1],[[2]], [[[3]]]]`, and so on.

   You can check whether the value of a variable, say `mylist`, is a list or not by the python expression `type(mylist) is list`; this evaluates to `True` if `mylist` holds a list, and `False` otherwise. Therefore you can write a conditional statement which controls the behavior of your program according to whether a certain object is a list or not.
5. Write a program that takes two sets from the user and prints the Cartesian product of the give sets in the form of a set of 2-tuples. You can use Python lists or sets to represent sets. [solution]

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;[13]
  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.

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

- 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)
```

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`.[14] 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:

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

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

- 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.

Exercise 9.1

1. Write a Python function no_111 that takes a string argument and returns True if the provided string is in the following language and False otherwise:

$$\{w \in \{0,1\}^* \mid w \text{ has no substring } 111\}$$

Your function should NOT use built-in string methods that search for a certain substring in a given string. Use a counter instead. [solution]

Note that although the question asks for a function rather than a program, you need to test your function for correctness. The easiest way to do it is to have a program that has your function definition and uses it. Below is an example that tests a string for whether it starts with a capital letter or not. There are lines starting with #. These comments are ignored by the Python interpreter. They are written for other programmers to help them understand your programs. Download the program to your computer, and run to observe its behavior.

```
#On the top part of the program, you have your function definition

def starts_with_cap(word):
    if word[0].isupper():
        return True
    else:
        return False

#The interpreter starts to execute your program from here on.
#It consults the definition of the function starts_with_cap
#when it is needed.

input_string = input("Enter a word ('X' for quit): ")

while input_string != 'X':

    if starts_with_cap(input_string):
        print('Starts with a capital letter.')
    else:
        print('Does not start with a capital letter.')

    input_string = input("Enter a word ('X' for quit): ")

print('Good-bye!')
```

2. Assume that exam grades are kept as a list of tuples. Each tuple pairs a student with her/his grade, e.g. [('John',85),('Mary',90),('Maya',54)]. Write a function that takes the student-grade list as input and returns the class average. Your function should work for any class size.
3. Write a python program that takes two lists of integers from the user and returns the number of elements that occur at the same position in both of the lists. **Do not use a function definition**. Note that, your program should be able to handle the case where the two lists are not equal in size. In reading a list from the user, put your input() function inside eval(); this way the user can directly type a list. Observe the following sample interaction:

```
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.
>>> my_list = eval(input('A list please: '))
A list please: [1,2,3]
>>> my_list
[1, 2, 3]
>>> type(my_list)
<class 'list'>
>>>
```

Some example sessions of your program would look like the following.

```
A list please: [1,[2,3],3,5,4]
A list please: [2,3,3,4,[3,4]]
the number of corresponding elements is:  1


A list please: [1,2,3,4,5,6,7]
A list please: [1,2,3]
the number of corresponding elements is:  3
```

4. Modify your program so that the number of corresponding elements is returned by a function.

5. Write a python program that takes a list as input, prints each member to the screen, together with the information whether the member has been encountered before or not. A sample session would be like the following.

```
A list please: [1,3,1,4,2,3]
1 new to me
3 new to me
1 seen before
4 new to me
2 new to me
3 seen before
```

10 Taking notes on your code

- Good computer code is usually self-explanatory. Using meaningful identifier names and having a neat and clear structure are good programming practices.

- However as programs get complicated, it becomes necessary to add some text to your code that is meant for a programmer – you, in the first place. The usual name for such devices is **comment**.

- Single-line comments are written after '#' character. Anything that comes after '#' it until a line break is ignored by the interpreter.

```
m = 4 #an integer assigned stored in a variable
n = 4 #here is another one

print(m*n) #integers multiplied, result printed
```

- If you need to use line breaks in your comment, Python provides a block comment structure. A block comment starts and ends with three consecutive double quotes and can include line breaks. Block comments can appear in anywhere in your code; but those immediately following a function definition appear on the help of that function (TODO: more on this later).

  Example 10.1
  Here is the solution to Exercise 8.2.5 put into function form:

```
def product(A,B):
    """Takes two sets A and B, and returns their
    cartesian product as a set of 2-tuples."""

    product = set()
    for x in A:
        for y in B:
            product.add((x,y))

    """Now it is time to return the result"""

    return product
```

□

11 Some basic programming techniques

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

  Example 11.1 (Count occurrences)

```
def occurs_count(lst,obj):
    """returns the number of times obj occurs in lst"""
    count = 0
    for x in lst:
        if x == obj:
            count += 1
    return count
```

□

- Note the way the `count` variable is incremented. It is a short form that we have been avoiding till now. It is equivalent to the statement '`count = count + 1`'.

11.2 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.

- At any given point of this computation we need to have access to two values. One is how many more m's we need to add; the other the value of m's added so far. We use a counter – n itself – for the first and an **accumulator**, `product`, for the second.

Example 11.2 (Multiplication)
A program that multiplies two integers without using a multiplication operator.[15]

```
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))
```

□

11.3 Flags

- Counters and accumulators are devices that are incremented or decremented as the computation advances. Their size depend on the input. Some programming tasks call for keeping track of the state of a variable, called a **flag**, which can be in one of a fixed number of states. If the number of states is 2, then a boolean type variable is used to store the state, like an on/off switch. Here is an example:

Example 11.3
Below is a function that takes a string and returns `True` if the input string consists of zero or more `'a'`s and `'b'`s without more than 3 consecutive a's, and returns `False` otherwise.[16]

```
def check(in_str):
    """Checks whether the string consists of a's and b's
    without more than 3 consecutive a's"""

    accept=True #flag for accept/reject status
    a_count = 0 #counter for cons a's

    for c in in_str:
        if c == 'a':
            a_count += 1
        elif c == 'b':
            a_count = 0
        else:
            #reject if any sym not a or b
            accept = False

        if a_count > 3:
            #reject if more than 3 cons a's
            accept = False

    return accept
```

□

[15] If you wonder how would this roundabout way of implementing multiplication compares with Python's multiplication operator '`*`' speed-wise, try multiplying 12 with 6666666666 (twelve 6's in a row), first directly on the Python interpreter, then by the program.

[16] This is not the optimum way to solve this problem; the function loops through the input string till the end even in cases a reject decision becomes inevitable at a non-final iteration. The purpose of the code is to illustrate the use of a flag.

11.4 Windows

- In all the examples so far, the decision of how to manipulate our counters and/or accumulators were based on the current item in a loop. In some programming tasks you need to check a number of items coming from consecutive iterations of a loop. An example:

Example 11.4
Now we have a slightly different string recognition task. Here we need to accept strings of zero or more a's and b's that do *not* contain the substring `'abab'`.

```
def check(in_str):
    """Checks whether the string consists of a's and b's
    without the pattern abab"""

    accept=True  #flag for accept/reject status
    window = ''  #keep track of last 4 symbols

    for c in in_str:
```

```
        if c == 'a' or c == 'b':
            #append c to the end of windo
            wwindow += c
            if len(window) > 4:
                #trim the window to 4 syms
                window = window[1:]
        else:
            #reject if any sym not a or b
            accept = False

        if window == 'abab':
            #reject if abab catched
            accept = False

    return accept
```

□

Exercise 11.5

1. Write a function named `myJoin` that takes a string type separator and a list of strings and returns a string where the strings in the list are joined by the separator.

2. Write a function named `compare` that takes two integers, returns `"GT"`, `"LT"` or `"EQ"`.

3. Write a function that takes a list and returns a list obtained by removing the first element of the input list. If the list is empty the function should return back the empty list. [solution]

4. Write a function that takes an input list and removes its first element. The input list needs to be one element less after the function is applied. [solution]

5. Write a function that takes a list of integers and an integer and removes all occurrences of the integer from the list and returns the modified list. [solution]

6. Write a function that takes a list of integers and an integer and removes only the first occurrence of the integer from the list and returns the modified list. [solution]

7. Take a third argument `n` and remove at most `n` occurrences. [solution]

8. Remove the last occurrence of an item from a list; do NOT reverse the list or traverse it from backwards.

9. Input: a list; Output: a palindrome of the list (e.g. given [1,2,3], returns [1,2,3,3,2,1]). Do not use builtin reverse function.

10. A program that takes a string as input and prints a horizontal histogram of the characters like below, for instance for the input `"abracadabra"`

    ```
    a *****
    b **
    c *
    d *
    r **
    ```

    [solution]

11. Make a vertical histogram. [solution]

Exercise 11.6

1. Write a program `fact` that gives the factorial of a given non-negative integer. (Factorial of 0 is 1, factorial of n is $1 \times 2 \times \ldots n$.)[solution1] [solution2]

2. Write a function that takes a list as input and returns its reverse. Use an accumulator in your solution. Also write functions that works for strings and works for both. [solution]

3. Write a function `flip` that takes a string and returns another string where the case of each character is flipped from upper to lower or vice versa. Check the string methods `isupper()`, `upper()`, `islower()`, and `upper()`–do not use `swap()`. [solution]

4. Write a function `seclarge` that takes a list of integers as input and returns the second largest element of the list. You need to write a function that applies to a list of integers of any length and there is no requirement that the elements of the lists be distinct. Make a list of all exceptional cases (empty list, lists with repeated elements, and the like), decide what to do in these cases, and make your function handle all these cases smoothly. [solution]

5. Write a function `swap` that takes a list and two indices as arguments and swaps the elements at the given indices. Your function should return the modified list, if swap is successful; otherwise – for instance if one or both of the indices is beyond the size of the list – your function should return the special python expression `None`. [solution]

6. Write a function (`maincon`) that takes a propositional logic formula as a string and returns its main connective. Assume that input formulae are totally parenthesized – no parenthesis omission by convention. Use `-`, `>`, `A`,`V`, for logical connectives $\neg$, $\rightarrow$, $\wedge$, $\vee$, respectively. Use lower case letters for proposition symbols. [solution]

12 Working with files

- To handle a file you need a **stream** object:

```
f = open("textfile.txt","r")
```

The file ‘textfile.txt’ needs to be in the directory you run the program or invoke the console.

- The second argument to the function `open()` is the **mode** of the stream object,[17] and the most common ones are `"r"` for read, `"w"` for write, and `"a"` for append; combinations are possible. Check here for additional modes.

- Once you have a stream object in read mode, the method `readlines()` returns the list of lines in the file. If you want to read lines one at a time, you need to use `readline()`. You can use the help function of the console to inspect other methods on streams.

Example 12.1
Let us read a list of Turkish words and compute the average word length.

```
# The file we will read has a single word in each line
# Therefore reading all the lines in a file into a list
# directly gives a word list. Words, however, come with
# a newline character '\n' at the end; to get rid of this
# python provides the method strip(). Observe its use below.

f = open("tr-word-list.txt", "r")

sum = 0
wlist = f.readlines()

# we no longer need the file, so we close the
# stream
f.close()

for w in wlist:
    sum += len(w.strip())

meanwl = sum/len(wlist)

print("The mean word length = ",str(meanwl))
```

- In order to write to a file, open a stream with `"w"` or `"a"` mode. The latter adds output to an already existing file, while the former creates a new file or overwrites what was already there. The basic writing method is `.write()`, you can also write a list of strings as lines by `.writelines()`.

Exercise 12.2

Read words as in Example12.1. Write palindromes to a separate file, word per line.

13 Importing modules

- By default, only a small number of basic functions and classes[18] are available for the Python interpreter. For instance there is no means to generate random numbers by default. Python 3 provides the **module** named `random` for this and related functionality. In order to be able to use functions in a Python module, you need to import the module:

```
>>> import random
```

- Now you can use functions from the module by prefixing their names by `random.`. Here is how to generate a random integer from a certain range, including the end points:

```
>>> random.randint(0,9)
2
```

- Other useful functions are `choice()`, `sample()` and `shuffle()`.

[17] Strictly speaking, it is a `TextIOWrapper` object.

[18] A class is a program that defines types of objects together with various methods of those objects.

```
>>> random.choice(["Alpha","Bravo","Charlie","Delta","Echo"])
'Charlie'
>>> random.sample(range(1,1000),10)
[855, 16, 433, 314, 140, 398, 126, 386, 862, 670]
>>> list = [1,2,3,4,5]
>>> random.shuffle(list)
>>> list
[2, 4, 1, 3, 5]
```

- Be careful that `shuffle()` does not return a value, it shuffles its list argument *in-place*.

Exercise 13.1

1. Read in the word list file as in 12.1. Import the module `statistics`. After importing the module you can check the contents by `help(statistics)` in the console or visit here. Find out how to compute mean and standard deviation. Use this methods to calculate the mean word length and standard deviation. Write in a separate file all the words that are longer than mean + 2 times standard deviation. This will give you ”unusually long” words of Turkish language.
2. Read in the word list; randomly pick a word; shuffle it, ask the user to guess the original, unshuffled word.
3. Modify your solution to the previous item, such that a correct answer need not be the original word, any word in the word list would do.
4. Write a function that takes a set of integers (you can take it as a list and eliminate repetitions) and generates a random binary relation as a set of ordered tuples.
5. Write functions that check functionhood, reflexivity, symmetry and transitivity of binary relations.

14 Higher order functions

- Higher order functions are functions that receive and/or deliver functions.

Example 14.1
The following function takes a function and a list, and applies the function to all the items in the list and returns the results, again in a list in the same order.

```
def lapply(f,l):

    retval = []
    for x in l:
        retval.append(f(x))

    return retval
```

□

Example 14.2
Now a function which takes a function `f`, a value `k` and the number `n` of times `f` is to be applied.

```
#for
def mapply_f(f,k,n):
    retval = k
    for x in range(0,n):
        retval = f(retval)
    return retval

#while
def mapply_w(f,k,n):
    retval = k
    while n != 0:
        retval = f(retval)
        n -= 1
    return retval

#recursive
def mapply(f,k,n):
    if n == 0:
        return k
    else:
        return mapply(f,f(k),n-1)
```

□

Example 14.3
Now a function whose return value is a function; taking two function arguments and returning their composition:

```
def compose(f,g):
    def retval(x):
        return f(g(x))
    return retval
```

Save this function definition in a file `ftools.py` (for "function tools"); you can load the definitions in a python file by importing it either in another program or in the console. For now, the file should be in the same directory with the importing program or where you invoked the console. And you need to prefix whatever you call from the loaded module with the module name.

```
>>> import ftools
>>> smax = ftools.compose(str,max)
>>> smax([100,102,1002])
'1002'
>>> s2max = ftools.compose(len,smax)
>>> s2max([100,102,1002])
4
```

□

15 Lambda abstraction

- In its most ordinary use **`lambda`**'s provide a concise way of defining functions. Here is a function that duplicates a string or a number:

```
>>> f = lambda x : 2*x
>>> f(4)
8
>>> f("vah")
'vahvah'
>>> f(f("vah"))
'vahvahvahvah'
```

- The **`lambda`** keyword is followed by the argument and the body of the function comes after the `:`.
- You do not need to name **`lambda`** functions in order to use them; you may directly use the definition itself, do not forget the parenthesis around the function and the argument:

```
>>> (lambda x: 3*x**2 + 5*x + 2)(8)
234
```

- You may have as many arguments as you like:

```
>>> (lambda x, y : (x+y)/(x*y))(2,4)
0.75
```

- The above function would give a division-by-zero error, if any of its arguments is zero. Therefore, it would be good to check for this and return `None` if this is the case:

```
>>> k = lambda x, y : (x+y)/(x*y) if x*y != 0 else None
>>> k(2,4)
0.75
>>> type(k(0,4))
<class 'NoneType'>
```

- Given a function with more than one arguments, you can abstract over any argument you like via **`lambda`**. Here is an example. Python's math module provides a logarithm function whose default base is $e = 2.78281828\ldots$.[19] In order to use it in another base you need to specify the base as a second argument:

```
>>> import math
>>> math.log(8)
2.0794415416798357
>>> math.log(8,2)
3.0
```

Assume you need to use the logarithm function in base 2 frequently. You can define your own logarithm function as:

```
>>> import math
>>> log = lambda x : math.log(x,2)
>>> log(8)
3.0
```

- You can also abstract over the base argument. The following function takes one argument and returns the logarithm of 8 taking the argument as the base:

[19] Logarithm function $\log_b a$ gives the exponent x such that $a = b^x$.

```
>>> import math
>>> f = lambda x : math.log(8,x)
>>> f(2)
3.0
>>> f(10)
0.9030899869919434
```

- Let us now go higher order and define a function that takes a two-place function and abstracts over the first argument of this function, specifying its second argument.

```
>>> abstract1 = lambda f, x : lambda y : f(y,x)
```

Note that the return value of the function `abstract1` is itself a function – the part coming after the first ':'. If you want to turn the Python's built-in logarithm function to a function operating at base 2, all you need is:

```
>>> import math
>>> log2 = abstract1(math.log,2)
>>> log2(16)
4.0
```

- Here is a similar function that abstracts over the second argument, specifying the first argument:

```
>>> abstract2 = lambda f, x : lambda y : f(x,y)
```

- Let us use `abstract2` on the binary function `math.pow` giving the result of raising its first argument to the power of its second argument.

```
>>> import math
>>> math.pow(2,3)
8.0
>>> pow2 = abstract2(math.pow,2)
>>> pow2(3) == math.pow(2,3)
True
```

- Generalizing one step further, we arrive at a function `abstract` that takes which argument – first or second – to abstract over as a parameter (1 for the first and 2 for the second):

```
>>> abstract = lambda f, x, n :\
... lambda y : f(x,y) if n==2 else lambda y : f(y,x)
>>> import math
>>> abstract(math.pow,10,2)(3)
1000.0
```

Note the use of the backslash character '\' to break lines

16 List comprehensions

- It is a highly common programming task to map a list of items to another list. For instance, you have a set of integers and you want the set of squares of these integers. One way to do it is to initialize an accumulator and collect all the squared integers in this accumulator through a for loop iterating over the original list. Python provides a concise way of realizing such processes.

```
>>> import math
>>> k = [1,2,3,4,5]
>>> [math.pow(x,2) for x in k]
[1.0, 4.0, 9.0, 16.0, 25.0]
```

- You can apply a filter to a list comprehension. For instance, if you want only the squares of even numbers in a list, you do:

```
>>> import math
>>> k = [1,2,3,4,5]
>>> [math.pow(x,2) for x in k if x%2 == 0]
[4.0, 16.0]
```

Exercise 16.1
Write a function `closest` that takes 3 arguments:

`x`: an integer;

`y`: an integer;

`k`: a list of unary functions on integers.

The function `closest` should return the value closest to `y` obtained by applying one of the functions in `k` to `x` once.

Exercise 16.2
Collatz Conjecture states that starting from any positive integer, one eventually arrives at 1, repeatedly applying the following function:

$$C(n) = \begin{cases} n/2, & \text{if n is even} \\ 3n+1, & \text{otherwise.} \end{cases}$$

For instance, the Collatz series for 6 would be, 6,3,10,5,16,8,4,2,1.

1. Write a function that computes the Collatz series for a given integer.
2. Write a function that takes an integer `n` and a list of integers `k`, and filters out elements in `k` which are NOT in the Collatz series of `n`.
3. Write a function that maps a list of integers to the list of number of steps required to reduce them to 1.
4. Write a function that takes two integers `x` and `y`, and returns the number of integers less than or equal to `x` that can be reduced to 1 in `y` or fewer steps.

---

17 Recursion

- An 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 ← 1
    count ← 0
    while count < n do
        count ← count + 1
        result ← result × 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 17.1

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

- 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 17.2

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
>>>
```

- 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.

Index