

Prac 1: “Python programming” (BIOL3014/BINF7000)

The aim of this first practical session is to give you key elements of the python language to be used during the following practicals, projects and lectures. The very basics of python are first reviewed and then the python libraries developed for this course will be shortly described. These libraries must be downloaded from <http://bioinf.scmb.uq.edu.au/pubsvn/binfpy/src/> (SVN repository). You should already be familiar with the python interpreter IDLE as well as how to make the libraries available to your programs (if not refer to appendix 2 of “Python Guide”).

There are seven exercises that you will have to complete. You will be mainly (not only) assessed on the python code and it is therefore essential that you paste your code into your report. **The maximal score for this prac is 5 marks.**

Variables and simple data types

A variable is the equivalent of a box in which you could put stuff in. The stuff can have different types, such as **integer**, **float**, **string** or other **class** objects. A variable can have any name (except the ones already used by python keywords), and in this document the variable will always start with "my" to help you differentiate them from keywords (e.g. myvar, myfile, mywhatever).

Integer	Floats	Strings
<pre>>>> myvar=12 >>> type(myvar) <type 'int'> >>> print myvar 12</pre>	<pre>>>> myvar = 12.2 >>> type(myvar) <type 'float'> >>> print myvar 12.2</pre>	<pre>>>> myvar = "hello" >>> type (myvar) <type 'str'> >>> print myvar "hello"</pre>

- Strings are defined explicitly by surrounding the text between double or simple quotes.
- "12" is not 12: "12" is the concatenation of the characters "1" and "2", but 12 is an integer !!!
- 11 is not 11.0, try typing 11/2 and 11.0/2 in the interpreter and observe the difference
- **type conversion**: `int("12")`, `float("12")`, `float(11)`

More on integers and floats

Note how to add and multiply the content of a variable and store the result in the same variable.

<pre>>>> myvar = 12 >>> myvar = myvar + 1 >>> print myvar 13 >>> myvar += 1 >>> print myvar 14</pre>	<pre>>>> myvar = 14 >>> myvar = myvar * 2 >>> print myvar 28 >>> myvar *= 2 >>> print myvar 56</pre>
---	---

More on strings

Strings have a range of useful predefined functions (that you could list using `help(str)`).

<pre>>>> myvar="ATCGTTTTCGATC" >>> print myvar.count("A") 2 >>> print len(A) 13 >>> print myvar.replace("T","U") "AUCGUUUUCGAUC" >>> print myvar "ATCGTTTTCGATC" >>> myvar = myvar.replace("T","U") >>> print myvar "AUCGUUUUCGAUC"</pre>	<pre>>>> print "U" in myvar True >>> print "T" in myvar False >>> print myvar[0] "A" >>> print myvar[1] "U" >>> print myvar[3] "C" >>> myvar += "AAAAAA" >>> print myvar "AUCGUUUUCGAUAAAAAA "</pre>
--	---

Lists and dictionaries

A variable can also contains other "boxes", with the two most useful types of box containers being the **lists** and **dictionaries**.

Lists: ordinal index	Dictionaries: alphanumerical keys																		
Delimiters: square brackets	Delimiters: curly brackets																		
<pre>>>> myvar = [1,2,12.2,"P23456"]</pre>	<pre>>>> myvar={"name":"hemoglobin", 100:"PRO2979", "length":147}</pre>																		
<table><tr><th>Index</th><th>Value</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>12.2</td></tr><tr><td>3</td><td>"P23456"</td></tr></table>	Index	Value	0	1	1	2	2	12.2	3	"P23456"	<table><tr><th>Key</th><th>Value</th></tr><tr><td>"name"</td><td>"hemoglobin"</td></tr><tr><td>100</td><td>"PRO2979"</td></tr><tr><td>"length"</td><td>147</td></tr></table>	Key	Value	"name"	"hemoglobin"	100	"PRO2979"	"length"	147
Index	Value																		
0	1																		
1	2																		
2	12.2																		
3	"P23456"																		
Key	Value																		
"name"	"hemoglobin"																		
100	"PRO2979"																		
"length"	147																		
<pre>>>> print type(myvar) <type 'list'> >>> print myvar[3] P23456 >>> myvar[3] = 15 >>> print myvar[3] 15 >>> myvar.append("PIP") >>> print myvar [1,2,12.2,15,'PIP'] >>> print myvar[1:3] [2,12.2] >>> print len(myvar) 5 >>> print "PIP" in myvar True >>> print "123456" in myvar False</pre>	<pre>>>> print type(myvar) <type 'dict'> >>> print myvar["name"] hemoglobin >>> print myvar[100] "PRO2979" >>> myvar["length"] = 120 >>> print myvar["length"] 120 >>> myvar["ID"] = "P69891" >>> print myvar["ID"] P69891 >>> print myvar.keys() ['length',100,'name','ID'] >>> print len(myvar) 4</pre>																		

filter: understand the keyword **filter**, which allows to select some elements of a list based on a condition. For example, if you would like to select all the elements of `myList1` that are also in `myList2` and put this common elements in `myList3`, you could use:

```
>>> myList3 = filter( lambda x: x in List2, List 1)
```

Control structures

`if` and `for` are the most common control structures used to execute instructions according to a certain conditions and repeatedly, respectively. The instructions that are under controlled are identified by their common level of indentations, which could be indicated by tabulations or blank spaces. We recommend in this course to always use blanks because it increases readability (especially for us to understand the code you will paste in your answers).

Conditional statements : *if*, *elif*, and *else*

The conditional operators are `==`, `>`, `<`, `>=`, `<=`, `is` and `in` (note that single `=` is not an operator). Any function returning boolean values (`True` or `False`) could also be used as a condition. Several conditions could be combined using `and` and `or` keywords.

Syntax	Example
<pre> if condition1: do something if condition1 is True do more things if condition1 is True do even more things if condition1 is True elif condition2: do something if condition2 is True and condition1 is False else: do something if condition1 and 2 are False do more things if condition is False do something whatever True or False </pre>	<pre> mycodon="ATG" if "T" in mycodon: print "The codon is DNA, changing to RNA" mycodon = mycodon.replace("T","U") else: print "The codon is possibly RNA" if mycodon == "AUG": print "This is the start codon!" print "The analysed codon is",mycodon </pre>

Note how blocks of instructions are defined by the number of blanks at the start of the line

Exercise 1 (½ mark)

Complete the following program so that it will test if a given amino acids one letter code is in the dictionary my_aacodes, which could be used to translate an amino one letter code to three letter codes. If it is then the program should print the three amino acid code, otherwise it should print an error message explaining that the given letter does not correspond to an amino acid. Please provide the code in your report (you will be marked on this code not on the output).

```

my_aa = "W"
my_aacodes = {'Y':'TYR', 'F':'PHE', 'C':'CYS', 'V':'VAL', 'H':'HIS', 'L':'LEU', 'D':'ASP',
'A':'ALA', 'S':'SER', 'E':'GLU', 'W':'TRP', 'P':'PRO', 'I':'ILE', 'R':'ARG', 'K':'LYS',
'N':'ASN', 'M':'MET', 'T':'THR', 'G':'GLY', 'Q':'GLN'}
...

```

The expected output of your code is:

```

The provided letter is W
This letter is in the list, it corresponds to the amino acids TRP

```

Now change the first line of your code into: my_aa = 'Z'. Running the code again should provide the following output:

```

The provided letter is Z
This letter is not the list and therefore does not correspond to an amino acids

```

Executing repeatedly for each element of a list: for, in

Syntax	Example
<pre> for variable in list: do something with variable do something else with variable do something out of the loop </pre>	<pre> myHydrophobics = "FLIMVPAWG" mySequence = ["M","H","K","L"] for myaa in mySequence: print "The amino acids is",myaa if myaa in myHydrophobics: print " It is hydrophobic" print "End of the program" </pre>

Do not forget the indentation !

Exercise 2 (1 mark):

2.1 In the example above, replace the second line by mySequence="MHKL". How is the output of the program changed? Explain.

2.2 Modify the code so that instead of printing if a residue is hydrophobic or not, a character "*" is printed for hydrophobic and "." for not hydrophobic.

will be added to a string variable called "mySequenceHydro" if the amino acids is hydrophobic and a blank " " if it is not hydrophobic. At the end of the program, print the amino acid sequence (mySequence) on one line, and on the following line the hydrophobic sequence (mySequenceHydro). Provide the code in your report. Expected output:

```
MHKL
* *
```

2.3 Modify the content of mySequence to be (copy/paste):

```
MCGTEGPNFYVPFSNKTGVVRSPFEAPQYYLAEPWQFSMLAAYMFLIMLGFPINFLTLYVTVQHKLRTPLNILLNLAVADLFMVFGGFTTLYTSL
HGYPVFGPTGCNLEGGFATLGGEIALWSLVLAIERVVVCKPMSNFRFGENHAIMGVAFTWVMALACAAPPLVGWSRYIPEGMCSCGIDYYTPHEET
NNESFVIYMFVVHFIIPILVIFFCYQQLVFTVKEAAAQQQESATTQKAEKEVTRMVIIMVIAFLICWLPYAGVAFYIFTHQGSFCGPIFMTIPAFFAKT
SAVYNPVIIYIMMNKQFRNCMVTTLCCKGNPLGDDEASTTVSKTETSQVAPA
```

Print the hydrophobic sequence of this sequence and identify hydrophobic clusters. How many hydrophobic clusters do you see? What kind of protein is this and why (biological insight) ?

Advanced topics:”

- the break statement allows to terminate a for loop prematurely, *i.e.* to jump after it
- looping over the keys of a dictionary using: “for myKey in myDict:”
- looping over the keys and values simultaneously using:

```
for myKey,myValue in myDict.items():
```

Executing repeatedly over several lists simultaneously: for

Lets imagine you have several lists of same length, which is the equivalent of a spreadsheet were the columns are stored in different lists. How to simultaneously list all the elements of each list?

Index	List 1	List 2	List 3	...
0	List1[0]	List2[0]	List3[0]	...
1	List1[1]	List2[1]	List3[1]	...
2	List1[2]	List2[2]	List3[2]	...
...

There are three main simple strategies that you could use to loop simultaneously over values in lists of same length. Lets take the example of two lists containing some protein UniProt codes and their sequence length, respectively:

```
myProtCode = ['013572','Q12346','013583','P32329','Q12303','P53057']
myProtLength = [173,211,123,569,508,165]
```

Strategy 1) Incrementing an index using range

```
for myIndex in range(len(myProtCode)):
    print myIndex,myProtCode[myIndex],myProtLength[myIndex]
```

Strategy 2) Using enumerate(list): return two values, the index and the value at this index

```
for myIndex,myCode in enumerate(myProtCode):
    print myIndex,myCode,myProtLength[myIndex]
```


Strategy 3) Using zip(list1,list2): concatenate the values of several list

```
for myCode,myLength in zip(myProtCode,myProtLength):
    print myCode,myLength
```

Functions

There are several good reason why you should organise your code by writing functions:

- it allows you to reuse your code at several different place in your code,
- it makes your code easier to understand, and
- it allows you to identify bugs more easily (and also by not duplicating code prevent new bugs).

Syntax	Example
<pre>def myFunction (param1, param2, ...): do something with the parameters do more things with the parameters return value</pre>  <pre>myvar = myFunction(10, 2e-3, "PE1290")</pre>	<pre>def myIsDNA (mySeq): myNonDNA = "QWERYUIOPSDFHJKLZXVBNM" for myAA in myNonDNA: if myAA in mySeq: return False return True print myIsDNA("GTTCGACCA")</pre>

A value (integer, float, string, object...) is send back from the function using the return statement. The return statement could happened at any point in the function definition and will terminate its execution (see the example above with two return statements, i.e. two possible ends to the function).

Exercise 3 (½ mark)

What is the purpose of the function `myIsDNA` defined above? What are the two possible outputs? Copy the code from the exercise above and comment what each line does.

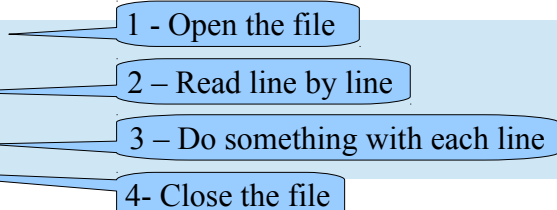
Reading, parsing and writing formatted text files.

A common task in Bioinformatics is to write scripts that parse data stored in formatted files. We will here gives the standard procedure to read and write files as well as some tips how to parse simply formatted files.

Reading text files

The syntax for opening text files, reading text files and writing into them is standard. You could therefore use the following examples as recipes.

```
myFile = open("filename")  
myIndex = 0  
for myLine in myFile:  
    myIndex += 1  
    print index, myLine,  
myFile.close()
```



The keyword `open` is used to open file and returns a file handler, which is stored in the variable `myFile`. The `for` loop then read each line of the file, and stores the each line in turn in the variable `myLine`. In the `for` loop the index of the line as well as the line itself. Finally, the file has to be closed.

Writing text files

Writing text data into files use a very similar syntax and use the function `.write()` of a filehandler object. For example, the following code write each value found in the list `myData` on a different line of a file named "myoutfile":

```
myData=["P1231","P234234","089098"]
myFile = open("filename.txt","w")
for myElement in myData:
    myFile.write(myElement+"\n")
myFile.close()
```

1 - Open a file for writing using the "w" option (write)

2 - Write the value in the file using .write(value)

3- Close the file (you might lose data if you forget to close it)

Note that the carriage return character "\n" (i.e. "go to the next line") has to be explicitly appended to each line of output. This is therefore a different behaviour than the command print that produce output on the screen. Note also that only strings could be passed to the function .write() and therefore numeric data will have to be converted into strings using str(myNumber).

Parsing text for each lines of a file.

The function split() of string objects are very useful for parsing simply formatted files in which data are into **columns** (i.e. data fields are organised in columns separated by a variable number of blanks) or if fields are **separated by tabulations**. An example of column formatted text:

```
1      kalata_B1      GLPVCGETCVGGTCNTPGCTCSWPVCTR
2      cycloviolacin_01 GIPCAESCVYIPCTVTALLGCSCSNRV
4      kalata_B2      GLPVCGETCFGGTCNTPGCCTWPICTRD
5      palicourein    GDPTFCGETCRVIPCTYSAAAGCTCDDRS
6      vhr1           GIPCAESCVWIPCTVTALLGCSCSNKVCYN
7      tricyclon_A     GGTIFDCGESCFGLGTCYTKGCSGEW
8      circulin_A      GIPCGESCVWIPCISSAALGCSCSNKVCYN
21     cycloviolacin_02 GIPCGESCVWIPCISSAALGCSCSNKVCYN
24     kalata_B6       GLPTCGETCFGGTCNTPGCSCSWPICTR
25     kalata_B3       GLPTCGETCFGGTCNTPGCTCDPWPICTR
26     kalata_B7       GLPVCGETCTLTGTCYTQGCTCSWPICR
27     cycloviolacin_08 GTLPCGESCVWIPCISSVVGCSCKSKVCYN
28     cycloviolacin_011 GTLPCGESCVWIPCISSAVVGCSCKSKVCYN
30     kalata_B4       GLPVCGETCVGGTCNTPGCTCSWPVCTR
31     vodo_M          GAPICGESCFGTGKCYTVQCS
32     cyclopsychotride_A SIPCGESCVFIPCTVTALLGCSCSKVCYN
33     cycloviolacin_H1 GIPCGESCVYIPCLTSAIGCSCKSKVCYN
...
```

You could use the following code to extract the data in each column:

```
myFile = open("cyclotide.txt")
for myLine in myFile:
    myFields = myLine.strip().split()
    print myFields[0], myFields[1], myFields[2]
myFile.close()
```

The line myFields = myLine.strip().split() do all the job of parsing each line. The function strip() remove the carriage return at the end of the line as well as the flanking blanks. The function split() then creates a list of elements for each column.

With the above example, parsing the first line will result in myFields[0] containing "1", myFields[1] containing "kalata_B1", and myFields[2] containing "GLPVCGETCVGGTCNTPGCTCSWPVCTR".

This simple technique will fail if the fields contain blanks (the .split() function will interpret it as several fields). If the different fields are separated by something else than blanks, for example tabulations, then the fields could be retrieved code by passing the delimiter to split(). For example, if the fields are separated by tabulations, e.g. (tabulations shown as →)

```
P8096HYR-Transcription factor from plant-1-21-MREVHLLLLLVL
P80IKDLC-Unkown protein-1-19-MDELVHLLLLM
...
```

You could simply modify the previous example to extract each line("\t" is the code for tabulation in Python):

```
myFile = open("proteins.txt")
for myLine in myFile:
    myFields = myLine.strip().split("\t")
    print myFields[1], int(myFields[3])-int(myFields[2])
myFile.close()
```

At the first instance of the for loop, myFields will contain ['P8096HYR','Transcription factor from plant','1','21','MREVHLLLLLVL']. Note how the second element contains blanks but was not split in several fields by .split(). Note also how the keyword int was used to convert strings into integer. Indeed, elements extracted by split() are of type string, and if you want to use the numerical value

that the extracted represent then you need to interpret the text into integer values or a float values.

Databases from EBI or NCBI do not provide data as tables but in flat files. For example a UniProt flat file looks like this:

```
ID E7KCZ5_YEASA Unreviewed; 254 AA.
AC E7KCZ5;
DT 05-APR-2011, integrated into UniProtKB/TrEMBL.
DT 05-APR-2011, sequence version 1.
DT 11-JUN-2014, entry version 8.
DE SubName: Full=YGR283C-like protein;
GN ORFNames=AWRI796_2020;
OS Saccharomyces cerevisiae (strain AWRI796) (Baker's yeast).
OC Eukaryota; Fungi; Dikarya; Ascomycota; Saccharomycotina;
OC Saccharomycetes; Saccharomycetales; Saccharomycetaceae; Saccharomyces.
OX NCBI_TaxID=764097;
RN [1]
RP NUCLEOTIDE SEQUENCE [LARGE SCALE GENOMIC DNA].
RC STRAIN=AWRI796;
RX PubMed=21304888; DOI=10.1371/journal.pgen.1001287;
RA Borneman A.R., Desany B.A., Riches D., Affourtit J.P., Forgan A.H.,
RA Pretorius I.S., Egholm M., Chambers P.J.;
RT "Whole-genome comparison reveals novel genetic elements that
RT characterize the genome of industrial strains of Saccharomyces
RT cerevisiae.";
RL PLoS Genet. 7:E1001287-E1001287(2011).
CC -!- CAUTION: The sequence shown here is derived from an
CC EMBL/GenBank/DBJ whole genome shotgun (WGS) entry which is
CC preliminary data.
CC -----
CC Copyrighted by the UniProt Consortium, see http://www.uniprot.org/terms
CC Distributed under the Creative Commons Attribution-NoDerivs License
CC -----
DR EMBL; ADVS01000026; EGA74818.1; -; Genomic_DNA.
DR OrthoDB; EOG7HB5KX; -.
DR Gene3D; 3.40.1280.10; -; 2.
DR InterPro; IPR029028; Alpha/beta_knot_MTases.
DR InterPro; IPR003750; Put_MeTrfase.
DR InterPro; IPR029026; tRNA_m1G_MTases_N.
DR PANTHER; PTHR12150; PTHR12150; 1.
DR Pfam; PF02598; MethyItrn_RNA_3; 1.
DR SUPFAM; SSF75217; SSF75217; 1.
PE 4: Predicted;
KW Complete proteome.
SQ
SEQUENCE 254 AA; 28828 MW; 2BC3529DC9D287D0 CRC64;
MKKKSRSKET ISDCLLLATL LQYFVTPPNL LDTTFKKKKNK LYLKCASTFP SLKQLPFMNA
SAEQHYKEGL SIARDSSKGGK SDDALTNLVY IGKNQIITLS NQNIPNTARV TVDTERKEVV
SPIDSYKGGP LGYHVRMAST LNEVSEGYTK IVWVNSGDFH YDEELSKYHK VETKLPYIAK
LKKSSSTSEKP CNILLIFGKW GHLKRCFRRS DLESSSLHHY FSGQLQFPAS IPQGNIPQID
SLPIALTMFQ RWAS
//
```

Accessing information from this type of files is slightly more complicated than from column formatted or tabulation separated text. This time the lines have to be tested for certain criteria that allows to pinpoint the lines of interest. For example to extract from the previous file the identifier, the organism, the molecular weight and the sequence, we could use the first two characters that stands as header to identify lines:

```
myID=""
myOrganism = ""
myWeight = 0
mySeq = ""
myFile = open("flatfile.txt")
for myLine in myFile:
    myFields = myLine.strip().split()
    if myLine[0:2] == "ID":
        myID=myFields[1]
    if myLine[0:2] == "OS":
        myOrganism=myLine[5:].strip()
    if myLine[0:2] == "SQ":
        myWeight = int(myFields[4])
    if myLine[0:2] == " ":
        mySeq += myLine.replace(" ", "").strip()
myFile.close()
print myID,myOrganism,myWeight,mySeq
```


Using python libraries.

Python libraries are files that contain definitions of functions and objects. These functions and objects can be used within your code by importing them with the `import` keyword. There are a very large number of available python libraries that you could use. Some of the standard python libraries you might want to have a look at are called `sys` (system), `os` (operating system) and `re` (regular expression). The UQ bioinformatics team has written a number of libraries that we will use during this course, and the main libraries that we will use this year are called `symbol`, `sequence`, `prob`, `stats`, `webservice` and `m1`. The following tables list some (not all) functions from the course libraries that we will use during the pracs, projects and lectures.

webservice: connect to EBI to retrieve some information through the Web	
Function	Description
<code>fetch(myID)</code>	Retrieves the sequence of a protein using its UniProt ID
<code>getGOTerms(myID)</code>	Retrieves the GO terms associated with one protein UniProt ID and returns them in a list
<code>getGOTerms([myID1,myID2,myID3])</code>	Retrieves the GO terms associated with several IDs
<code>getGODef(myGOTerm)</code>	Retrieves the definition of a GO term
<code>getGenes(myGOTerms,taxo=MyTaxonomyNumber)</code>	Retrieves all the proteins UniProt ID from an organism with a given taxonomy and GO terms

symbol: defines the Alphabet class used to define the symbols for proteins and nucleic acids	
Function	Description
<code>Alphabet()</code>	Class defining the objects of type Alphabet()
<code>DNA_Alphabet</code>	Alphabet for DNA sequences
<code>Protein_Alphabet</code>	Alphabet for protein sequences
<code>DSSP_Alphabet</code>	Alphabet for secondary structure sequences

sequence: class and functions to manage biological sequences	
Function	Description
<code>Sequence(mySequence,myAlphabet)</code>	Stores a biological sequence and its Alphabet
<code>readFastaFile(myFilename)</code>	Read a fasta format sequence files
<code>writeFastaFile(myFilename,mySequences)</code>	Write sequences in a file
<code>alignGlobal(mySequenceA,mySequenceB,mySubstMatrx)</code>	Global alignment between two sequences
<code>PWM(myForeground,myBackground)</code>	Defines a position weight matrix

prob: function to define simple probability distributions	
Function	Description
<code>Distrib</code>	A class for a discrete probability distribution, defined over a specified "Alphabet"
<code>Joint</code>	A joint probability class
<code>IndepJoint</code>	A joint probability of independent distributions
<code>NaiveBayes</code>	Naive Bayes classifier

ml: machine learning library for neural network and K means clustering	
Function	Description
NN	A class to define and use neural networks
readNNFile	Reads neural networks created with NN
Qk	Computes k accuracy
Kmeans	A class to perform K means clustering

stats: statistical functions	
Function	Description
getFETpval (a1,a2,b1,b2)	Fisher's exact test one tail
getFET2pval (a1,a2,b1,b2)	Fisher's exact test two tails
getPearson(X,Y)	Pearson correlation coefficient

You do not have to know all of these functions for now but you might want to investigate some of them on your own time. One way to learn how to use them is through the `help` command. To access the general help on a library (for example `webservice`) or to a specific function in the library (for example `getG0Def` from `webservice`) you could type in the python interpreter the following lines:

```
>>> import webservice
>>> help(webservice)
>>> help(webservice.getG0Def)
```

Within your python codes it will be easier for you to use a different syntax that allows to import directly the functions in the main name space of python, *i.e.* you could use the function right away as if you typed them yourself. As an illustration of this syntax:

```
from webservice import *
print getG0Def('GO:0005634')
myTerms = getG0Terms('P38903')
print myTerms
for myT in myTerms:
    print getG0Def(myT)
```

Please test the preceding small program for yourself.

We will now do a series of small exercises in which you will be asked to implement four python different programs that will have the same outputs. These exercises aim at giving you an idea of different programming styles and also to appreciate their impact on running time. The file “`yeast_transcriptome.txt`” contains information a Baker's yeast (*Saccharomyces cerevisiae*) proteome. We will analyse this proteome in terms of number of proteins that are in the nucleus or in the cytoplasm and the number of proteins that are transcription factors or not.

Exercise 4 (1 mark): first implementation

We will here use a subset of only 10 proteins from the 6621 proteins in the yeast transcriptome (the running time would be too long for the whole set of proteins). Please implement the different steps described in 4.1 to 4.4 and provide your code in the report as well as the output. To give you an idea, this program should run in less than 2 min on your computer.

4.1 Look at the content of the file “`yeast_transcriptome_10.txt`”; notice that the columns are separated by tabulations; identify the column with the UniProt identifiers (on the first line it is 'P38903'). Write a python code that opens this file, extracts the protein identifiers, and stores them in a list.

4.2 For each protein, use its identifier to retrieve the list of GO terms.

4.3 For each protein, identify if the GO term for nucleus (GO:0005634) is in the GO term list, and/or if the GO terms for TF (GO:0003700) is in the GO term list. Increment consequently the

content of a variable that acts as a counter for the number of TF in the nucleus, TF not in the nucleus, non-TF in the nucleus, non-TF not in the nucleus.

4.4 Compute the probability of a protein to be a TF and in the nucleus. Compute also the probability of being in the nucleus if we know that the protein is a TF.

Tips:

- 4.2 and 4.3 could be easily implemented within the same loop
- You probably used integer to count the number of proteins, whereas here you will need to use a float, so you should convert the number of proteins into floating values using `float()` before computing properties.

Exercise 5 (1 mark): second implementation

We will here use a subset of 100 proteins from the 6621 proteins. Please implement the different steps and provide your code in the report as well as the outputs. This program should run in less than 1 min on your computer.

5.1 Look at the content of the file “yeast_transcriptome_100.txt”, notice that it has same structure as the file used in 4.1. Write a python code that opens this file, extracts the protein identifiers, and stores them in a list.

5.2 Pass the whole list to the function `getGOTerms`, which will now return a dictionary with keys being gene names and values being list of GO Terms for each gene (note that the output is different depending if you pass a single GO Term or a list of GO terms to `getGOTerms`).

5.3 For each gene name in the dictionary, increment one of the four counter variables (TF in nucleus, TF not in nucleus, ...) depending of the protein is a TF and/or in the nucleus, as in 4.3.

5.4 Compute the same statistics as in 4.4

Exercise 6 (½ mark): Third implementation

Using the file “yeast_transcriptome_100.txt”, modify your previous implementation of question 5.3 so that you will not use any conditional testing and not use counter variables. To do so use the keyword `filter` on the output of `getGOTerms` to establish the list of all proteins that are TF and then the list of all proteins that are in the nucleus. Use again `filter` to get the list of all the proteins that are TF and in the nucleus. Knowing the total number of proteins, you could easily compute the number of TF proteins in the nucleus, non-TF protein in the nucleus, TF not in the nucleus and non-TF not in the nucleus.

Tip: Note that if a protein does not have GO terms, then it will not have an entry in the dictionary returned by `getGOTerms`. These proteins not in the dictionary will be considered as being not in the nucleus and not being TF.

Exercise 7 (½ mark): Fourth implementation

Could you rewrite this code using the function `getGenes` (and not `getGOTerms`) knowing that the taxon identifier for *Saccharomyces cerevisiae* is 559292? This code should run considerably faster and you will use the full transcriptome in “yeast_transcriptome.txt” (running time <1 min).

Note that `getGenes` will return all the sequences in UniProt that match your query and not only the ones from the transcriptome experiment. You will therefore have to select only the proteins that are described in “yeast_transcriptome.txt” among all the ones returned by `getGenes`