

# A Python Reference Guide for Bioinformatics @ UQ

## Introduction

Python is a popular programming language that is both easy to learn and readable. This guide aims to help you understand Python concepts that come up as you start using programming.

In this guide, we use examples that are inspired by bioinformatics problems; most of these examples are available in the “guide.py” Python file that you can download from the course Blackboard site. You should feel free to change that file and run it to get a feel for Python.

## More Helpful Resources

There are many resources from which you can get further help with Python. There is a multitude of books available, and a huge number of resources on the web. Just search for whatever you want to know or are stuck with; you will probably find an answer.

The official Python website contains a very good introductory tutorial, at:

<http://docs.python.org/tutorial/>

Topics in this guide also link to the corresponding section in the official Python tutorial, and it is a good idea to read the linked sections as you progress through.

Occasionally we are introducing something complicated that you may not fully understand at that time but could prove very useful. We have marked those paragraphs with a ☆.

## Using Python

<http://docs.python.org/tutorial/interpreter.html>

There are two main ways to use Python. The first is through interacting with the interpreter. When you open **IDLE** (or if you run the command "python" from a terminal), you get a prompt:

```
>>>
```

If you type a line of Python code, for example:

```
>>> print 'Hello world!'
```

and press `enter`, Python will *interpret* and *execute* that line of code. (Can you guess what will happen?) The interpreter is very useful for trying out ideas and testing things, as it is quick and easy to start and then *interact* with, in your own pace.

The other way to use Python is by writing code in a file with the `.py` extension, and *running* that file. You can do this by opening it in **IDLE** and selecting **Run Module** from the **Run** menu, or by using the command shell to type “python filename.py”.

Python files ending with `.py` are referred to as “modules” as they package variables, functions and classes that belong together. Running a module means that Python will

*load* the entire file (that means: *defining* methods/functions, classes etc., and *executing* statements that are outside definitions).

**Any code you want to keep for later or run again, you should save in *.py* files.**

From the interpreter (or *another* module) you can *import .py* files to interact with them. Importing a file or module essentially means that you *run* the module so that everything defined in it is available for you to use. Most of the examples below assume you have an interpreter open but in most cases they could also form part of a module.

There is a step-by-step guide to get IDLE up-and-running in Appendix 2.

## Why program?

We write programs to make computers do things for us.

Mostly, “doing things” equates to repetitive and complicated tasks involving math and logic. In other words, do things that humans aren't great at, and that computers can be instructed to do.

The following sections cover everything from the basic stuff you need to know to make anything happen, through to logical decisions and repetitive tasks, finishing up at tools like functions and classes that make the complexity of programming more manageable.

You're not expected to know everything in here straight away. Read through and try to understand as much as you can. It will start to make more sense as you use it.

## The Basics

<http://docs.python.org/tutorial/introduction.html>

### Comments

Comment lines in Python start with a hash (#), and continue until the end of the line. Text surrounded by triple single-quotes or triple double-quotes (e.g., `'''comment'''` and `"""comment"""`) are comments that can extend across multiple lines. Comments are great for documenting how your code works in plain English.

```
# This is a single line comment
''' This is a longer
    multi-line comment.
'''
```

In this document, whenever you see lines like this, they're just helpful comments from us to you.

### Arithmetic operations and operators

From the interpreter, Python can be used as a calculator. Addition (+), subtraction (−), multiplication (\*), division (/) and exponentiation (\*\*) are all available, and brackets

can be used to group operations, as you would expect. The percent sign (%) returns the remainder when you divide (i.e., the modulus).

```
>>> (50 * 3) + 6**2
186
>>> 5 / 3
1
>>> 5.0 / 3
1.666666666666667
>>> 10 % 3
1
```

Python has two kinds of *numbers*: integers and floats. Integers are whole numbers and floats are decimal numbers. Note that, e.g., 1.0 is a float, whereas 1 is an integer. In Python, division of integers results in an integer, so that 5/3 gives 1. If a float is involved in an operation, the result is a float, so we can get the expected answer by writing 5.0/3, which gives 1.6667.

The `int` and `float` functions can be used to convert values to different types, so that `float(5)/3` is the same as `5.0/3`, and `int(3.6)` is 3.

## Comparisons and Booleans

Python compares values using the `==` (equality), the familiar `>=`, `>`, `<=` and `<` (greater than or equal to, greater than, etc.) and `!=` (not equal) operators. The result of a comparison is a Boolean value, either `True` or `False`. You can chain together comparisons using the `not`, `and` and `or` operators, which works as you'd expect.

```
>>> 1 == 2
False
>>> not 1 == 1
False
>>> 5 > -2 or 1 > 4
True
>>> some_dna = 'AGCCT' # we assign a value (a string)
>>> some_dna != 'AGCCT' # we evaluate an expression
False
```

## Variables

The equals sign in Python is used to store values in variables (“assignment”). Variables must be defined before being used.

```
>>> a = 50
>>> b = 6
>>> (a * 3) + b**2
186
```

Note that two equals signs checks if two values or variables are equal, while one equals sign assigns a value to a variable. Getting this wrong will be very confusing, and can be difficult to notice.

## Strings

Text in Python is stored in *strings*. We will also represent biological sequences such as DNA, RNA and proteins as strings. Strings are a sequence of characters defined using quote marks (single *or* double—as long as you are consistent):

```
>>> dnaseq = "AGCCGGT"
```

Individual characters in a string can be accessed using square brackets, giving the position of the character you want. Note that computers tend to count from zero, so in order to get the first character in a string, you need to ask for position 0, not position 1. You can also ask for the last, second-last, etc. characters in a string using negative positions. The final letter in a string is at position -1 (not -0!).

```
>>> dnaseq[0]
'A'
>>> dnaseq[-1]
'T'
```

You can also work with *slices* in a similar way, giving positions as `start:stop`, which will return elements from `start` to one before `stop`, with an optional `:step`.

```
>>> dnaseq[0:2]
'AG'
>>> dnaseq[0:7:2]
'ACGT'
>>> dnaseq[::-1]
'TGGCCGA'
```

Slices are explored in more detail at:

<http://docs.python.org/tutorial/introduction.html#strings>

You can join strings together with the `+` operator.

```
>>> "My palindrome DNA sequence: " + dnaseq + dnaseq[::-1]
'My palindrome DNA sequence: AGCCGGTGGCCGA'
```

There are many functions available for strings, such as `len` that returns the length of the string and `replace` that replaces all instances of a particular substring with another one.

```
>>> dnaseq = "AGCCGGT"
>>> len(dnaseq)
7
>>> rnaseq = dnaseq.replace("T", "U")
>>> rnaseq
'AGCCGGU'
```

A list of all string functions is available at:

<http://docs.python.org/library/stdtypes.html#string-methods>

## Printing messages

The `print` command can be used to display strings as well as other variables. Use a comma to separate multiple variables and strings you wish to print on the same line.

```
>>> print "Hello world!"
Hello world!
```

```
>>> a=2
>>> print "The number two:", a
The number two: 2
```

## Lists (and tuples)

Lists are similar to strings, but instead of being a sequence of *characters*, they are a sequence of (any) *objects*. Lists are defined using square brackets around a comma-separated list of items. Accessing items in the list uses the same square bracket notation as strings. Note that the use of *indices* and *slices* is identical to that of strings.

```
>>> dnaseq = "AGCCGGT"
>>> someSequences = ["AACGTCG", "CGCGCTAT", dnaseq]
>>> someSequences[1:3]
['CGCGCTAT', 'AGCCGGT']
```

You can add an item to the end of the list using the `append` function.

```
>>> someSequences.append("AGGCCCTG")
>>> someSequences[3]
'AGGCCCTG'
```

Lists also have the `len` function to get their length, as well as a host of other functions. For a list and explanation of them, see:

<http://docs.python.org/tutorial/datastructures.html#more-on-lists>.

Tuples can be understood as *immutable* (unchangeable) lists. They are defined and accessed similarly but note the use of normal brackets:

```
>>> dna_alpha_tuple=('A', 'C', 'G', 'T')
>>> dna_alpha_list=['A', 'C', 'G', 'T']
>>> dna_alpha_tuple[2:4]
('G', 'T')
>>> dna_alpha_list[3] = 'U' # lists can change
>>> dna_alpha_list[2:4]
['G', 'U']
>>> dna_alpha_tuple[3] = 'U' # tuples cannot change!
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    dna_alpha_tuple[3]='U'
TypeError: 'tuple' object does not support item assignment
```

## Flow Control

<http://docs.python.org/tutorial/controlflow.html>

### If Statements

Controlling what happens in different situations is what makes programming so useful. The simplest way to do this is the `if` statement. You can make Python execute a section of code only *if* some condition is met. The `elif` (else if) and `else` statements can be used to include other possibilities.

Python uses indents (four spaces at the start of a line) to group code into sections. Note that the dots are automatically added by the standard Python interpreter (not

IDLE) to signal that it is expecting commands that should take place within the `if` clause—if you want to finish it press enter.

```
>>> a = 4
>>> if a > 3:
...     print "Greater than 3."
...     print "Yay!"
... elif a == 3:
...     print "Equal to 3."
... else:
...     print "Less than 3."
...
Greater than 3.
Yay!
```

Note that there are three blocks here; indenting starts a block, unindenting finishes it. The condition used in an `if` or `elif` statement can be anything that results in a Boolean value. Often, it is one of the comparison operators we saw before.

## While Loops

Repeating code many times is a common task, and this is done with loops. Python has two types of loops. The first is the “`while`” loop. The code inside the `while` loop (again, everything that is indented) will continue being executed *while* the loop condition is true. See the example below.

```
>>> i = 0
>>> while i < 5:
...     print i
...     i = i + 1
...
0
1
2
3
4
```

## For Loops

Python has another kind of loop, which is more common and will be used extensively later, namely the “`for`” loop. The `for` loop takes a sequence (like a string or a list), assigns each item in the sequence to the specified variable in turn, and executes the code inside the loop (the indented block) for each item.

```
>>> dna = 'ACGCAAGCGTGG'
>>> cnt = 0;
>>> for base in dna:
...     if base == 'A':
...         print 'Found adenine.'
...         cnt = cnt + 1
...
Found adenine.
Found adenine.
Found adenine.
```

```
>>> print 'Found %d adenines' % (cnt)
Found 3 adenines
>>> bases = ['A', 'G', 'C', 'T']
>>> for base in bases:
...     print base
...
A
G
C
T
```

Let's use `bases` above to check if a given string `seq` is a proper DNA string

```
>>> seq = 'ACGCAAGCGTGG'
>>> seq_proper = True # assume that the sequence is DNA
>>> for sym in seq:
>>>     sym_proper = False # assume the symbol is NOT a base
>>>     for base in bases:
...         if (sym == base):
...             sym_proper = True # we've seen the symbol is ok
...         if not sym_proper:
...             seq_proper = False
...
>>> print seq_proper
True
```

⊛ It is tedious to rewrite the whole test for a different string. Indeed it would be nice if we could perform the whole thing above with just a line of code like this:

```
>>> print validseq(bases, 'QCGCXZGTGG')
False
```

You would get an error if you tried that without first putting your code into a *function*—here's a function definition:

```
>>> def validseq(my_bases, my_seq):
...     seq_proper=True
...     for sym in my_seq:
...         sym_proper = False
...         for base in my_bases:
...             if (sym == base):
...                 sym_proper = True
...         if not sym_proper:
...             seq_proper = False
...     return seq_proper
...
>>> print validseq('ACGT', 'ACGCAAGCGTGG')
True
>>> print validseq('ACGT', 'UUCCGCGG')
False
>>> print validseq('ACGU', 'UUCCGCGG')
True
```

Now you can easily check any sequence, for any alphabet without repeating the code. (You will no longer get an error if you run the same line as before.) You have already

seen functions that are system-specified (e.g. `len('abc')` in the section about strings above, or the `replace` and `append` functions that come with the string `class`). We will discuss how to define more of your own later using the `def` keyword and how they can be associated with classes, but for now just note how handy they are in making your code modular.

One final point about loops: The `while` loop from the example above can be written using a `for` loop instead, using the `range` function. The `range` function returns a list of numbers spanning the range specified, which works the same way as the square bracket notation for strings, lists and tuples.

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(3, 8)
[3, 4, 5, 6, 7]
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
>>> for i in range(5):
...     print i
...
0
1
2
3
4
```

## Types of Variables

Python has many types of variables depending on what type of data you want to store. So far we've seen integers, floats, strings and lists. There are a few more types we will have use for in bioinformatics.

### Arrays

Much to the mathematician's horror, Python only comes with lists, which are not real arrays. While lists are easy for storing and accessing one-dimensional data, they become quite convoluted for two- or high-dimensional arrays (lists of lists of lists of...). Fortunately, the “numpy” package provides flexible, fast arrays for Python. You can use these arrays by typing:

```
>>> import numpy
```

into the interpreter, or at the top of your Python file. If this fails, you may need to install numpy on your computer from <http://www.numpy.org>. Arrays are used in SCIE1000, so you may already be familiar with them. In this course we will be using them to store matrices. For more information on numpy arrays, see:

<http://docs.scipy.org/doc/numpy/reference/arrays.html>.

You can create a two-dimensional array using the `numpy.zeros((n,m))` function, which creates an `n` by `m` matrix full of zeros (note the two lots of brackets). You can then access and set elements of the array using the square bracket notation we used for lists, except that for matrices we need to specify two numbers – the row and



column – separated by a comma. (The inner bracket-pair represents an  $n$ -element tuple, with one element for each dimension of your  $n$ -dimensional array.)

```
>>> import numpy
>>> matrix = numpy.zeros((4,5))
>>> matrix
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> matrix[1,2] = 3
>>> print matrix
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  3.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
```

## Dictionaries (maps)

Dictionaries are another type of collection of objects. A dictionary in Python is a collection of key-value pairs: a key, which cannot be changed and can be (among other things) a string or number, and a corresponding value, which can be of any type. Keys must be unique within a dictionary (i.e., you can't have two items with the same key). Items can be “looked up” in the dictionary by specifying a key and getting the corresponding value in return. Dictionaries are specified using curly brackets.

```
>>> codons = {"start": "ATG", "stop": "TAG"}
>>> codons["start"]
'ATG'
>>> dna = "ATGGTACAGGTAG"
>>> dna.find(codons["stop"])
10
>>> for name in codons:
...     print name
start
stop
```

This example used the `find` function of strings, which returns the index of the first place it finds the given substring. So the first stop codon in the DNA sequence is at index 10. That is, `dna[10:13]` is a stop codon.

You can loop through all the keys and values in a dictionary using the `items` function. Note that the order in which the key-value pairs are given is not determined by the order in which they were specified.

```
>>> d = {'A': 4, 'B': 2, 'C': 3}
>>> for key, value in d.items():
...     print key, 'has number', value
A has number 4
C has number 3
B has number 2
```

★ There will be times when you would like to sort entries... the function `sorted` does this on any iterable data type (e.g. lists and dictionaries).

```

>>> bases = ['A', 'G', 'C', 'T']
>>> sorted(bases, reverse=True)
['T', 'G', 'C', 'A']
>>> sorted(d) # the dictionary defined previously
['A', 'B', 'C']
>>> # Now for a nasty but useful operation...
>>> sorted(d.items(), key=lambda v: v[1], reverse=True)
[('A', 5), ('C', 3), ('B', 2)]

```

For more information on dictionaries, see:

<http://docs.python.org/tutorial/datastructures.html#dictionaries>

## Functions

Functions are sections of code that take some input (called arguments), perform some operations and then give output (called the return value). For example, the `find` function of a string takes a string to search for as an argument, and returns an integer indicating where and whether that string was found.

### Standard library

There are a number of functions available in Python only when they are `imported`, as we saw with numpy arrays. Another example of functions you will use that require importing are those provided by the `math` module.

```

>>> import math
>>> math.sqrt(9)
3.0
>>> math.pi
3.1415926535897931
>>> math.cos(0)
1.0

```

### User-defined functions

It is also possible to define your own functions. (We sneak-peeked at one `validseq` earlier.) The `def` statement is used to define a function by its name and input arguments. The `return` statement is used to make the function give output. Here is an example function that counts the number of adenine (A) bases in a given DNA sequence:

```

>>> def countAdenine(dna):
...     numOfAs = dna.count('A')
...     return numOfAs
...
>>> countAdenine('ACGCAAGCGTGG')
3

```

Note that when the function is defined, Python doesn't execute the code within it. *Running* this code only means *defining* it. *Execution* only happens when the function is used, which we do the same way as all the functions we have seen previously.

In some cases we want to package code that takes some arguments but not return any result. Then we simply omit the return statement—the function knows when to end because of the way indentation works in Python. People refer to such functions as *methods* or *procedures*. (You may wonder if methods are useful at all? For instance we might change a couple of variables that are accessible outside the method.)

For more information on defining functions, see

<http://docs.python.org/tutorial/controlflow.html#defining-functions>.

## Classes

Several of the examples in this guide have used DNA sequences, performing operations like finding certain codons and counting the number of the bases that make up the sequence. Wouldn't it be nice if Python had a special type of variable just for DNA sequences? Classes allow you to construct your own variable types, with their own functions and internal variables.

You can define a class using the `class` statement in a similar way to the `def` statement for functions. Classes define a template for a kind of *object* you wish to create. For a single class definition, like that of a DNA sequence, you can create many objects (that is, many DNA sequence objects). Let's define a simple sequence class.

```
>>> class DNASequence():
...     def __init__(self, sequence):
...         self.sequence = sequence
...     def getLength(self):
...         return len(self.sequence)
...     def countAdenine(self):
...         return self.sequence.count('A')
... 
```

The class is made up of a number of functions, all of which take `self` as their first argument. Inside these functions, the `self` variable refers to the object itself. We can use this reference to store variables inside the object; in the `__init__` function above, we store the given sequence inside the object for other functions to use later.

The `__init__` function is special; it is the function that is run when a new Sequence object is created. Let's create some Sequence objects to see how this works.

```
>>> seqA = DNASequence('AGCCCGGAT')
>>> seqA.sequence
'AGCCCGGAT'
>>> seqA.getLength()
9
>>> seqA.countAdenine()
2
>>> print "The sequence", seqA.sequence, "is", \
(float(seqA.countAdenine())/seqA.getLength())*100, "% adenine."
The sequence AGCGGCCT is 12.5 % adenine.
```

In the first line when the object is created, the `__init__` function is called with the given string as its sequence argument. It stores this inside the object, and we can access that variable as done in the next line. The two functions can then be used to

find out things about the sequence. We could define more useful functions (e.g., counting bases, transcribing the sequence to RNA, aligning the sequence with another, etc.), and in the first few practicals of this course, you will do exactly this.

More information on classes can be found at:

<http://docs.python.org/tutorial/classes.html#a-first-look-at-classes>.

The concepts of classes and objects are part of a style of programming known as object-oriented programming (OOP). In SCIE1000, a simple program might define a number of functions and then perform a list of tasks on data using those functions. In contrast, using OOP a program is thought of as a group of objects created from class definitions (as we saw above), interacting with each other using their own functions and variables. While other OOP concepts are not covered explicitly in this guide, we will use classes and objects, as discussed above, extensively.

## Files

Files are used to permanently store data. Many files but not all are text files, meaning that it is possible to view and edit them using a text editor, e.g. **TextWrangler** on Macs, **Notepad** on Windows PCs and **emacs** on Linux machines. To increase portability and readability, text files are often used for storing biological data. The information in them usually follows some pre-defined template or format. For example, biological sequence data is often stored in FASTA files (use a “.fa” or “.fasta” file extension).

### DNA and protein sequence data

Here’s an excerpt from an example FASTA file.

```
>P99029 PRDX5_MOUSE Peroxiredoxin-5, mitochondrial
MLQLGLRVLGCKASSVLRASSTCLAGRAGRKEAGWECGGARSFSSSAVTMAPIKVGDAIPS
VEVFEGEPGKKVNLAELFKGKKGVLFGVPGAFTPGCSKTHLPGFVEQAGALKAKGAQVVA
CLSVNDVFVIEEWGRAHQAEQKVRLLADPTGAFGKATDLLLDDSLVSLFGNRRRLKRFSMV
IDNGIVKALNVEPDGTGLTCSLAPNILSQL
>Q8QZX5 LSM10_MOUSE U7 snRNA-associated Sm-like protein
MALSHSVKERTISENSLIILLQGLQGQIITVDLRDESVARGRIDNVDAFMNIRLANVTTYT
DRWGHQVELDDLFVTGRNVRYVHIPDGVDITATIEQQQLQIIHRVRNFGGKGQGRREFPSK
RP
>P56959 FUS_MOUSE RNA-binding protein FUS OS=Mus musculus
MASNDYTQQATQSYGAYPTQPGQGYQQSSQPYGQQSYSGYGQSADTSGYGQSSYGSSYG
...
```

A FASTA file thus contain a “define” that always starts with a ‘>’ then the name of the sequence followed on the same line by other information. The lines before the next ‘>’ or end-of-file is the sequence data. The symbols used depend on what type of sequence. The following code consists of two methods where one calls the other and read a file on the FASTA format. It creates a sequence object for each entry in a specified file. (The definition of the used Sequence class is provided in the next section of this guide.) For now you can assume we’ve taken care of those details similar to how we dealt with a DNASequence above.

```
def readFastaString(string, alphabet):
    """ Read the given string as FASTA formatted data and return
```

```

    the list of sequences contained within it. """
    seqlist = [] # list of sequences contained in the string
    seqname = None # name of *current* sequence
    seqdata = [] # sequence data for *current* sequence
    for line in string.splitlines(): # read every line
        if len(line) == 0: # ignore empty lines
            continue
        if line[0] == '>': # start of new sequence
            if seqname: # check if we've got one current
                current = Sequence(seqdata, alphabet, seqname)
                seqlist.append(current)
            # now collect data about the new sequence
            seqname = line[1:].split()[0] # skip first char
            seqdata = []
        else: # we assume this is (more) data for current
            cleanline = line.split()
            for thisline in cleanline:
                seqdata.extend(tuple(thisline.strip('*')))
    # we're done reading the file, but the last sequence remains
    if seqname:
        lastseq = Sequence(seqdata, alphabet, seqname)
        seqlist.append(lastseq)
    return seqlist

def readFastaFile(filename, alphabet):
    """ Read the given FASTA formatted file and return the list
        of sequences contained within it. """
    fh = open(filename)
    data = fh.read()
    fh.close()
    seqlist = readFastaString(data, alphabet)
    return seqlist

```

Most biological data repositories provide their sequence data in the FASTA format. For example, you can download protein sequences from UniProtKB (<http://www.uniprot.org>) or PDB (<http://www.pdb.org/>) and nucleotide sequences from NCBI's GenBank (<http://www.ncbi.nlm.nih.gov/genbank/>) or UCSC's Genome Browser (<http://genome.ucsc.edu/>). In most cases, you can find the same sequence data in different repositories, often complemented with some special data (e.g. protein structure in PDB).

## Genome sequence

Genomes are very large so when we deal with the genomic sequence we need to be very mindful of efficiency and storage requirements. UCSC (mentioned above) provides so-called “2bit” files of the most popular genomes (<http://hgdownload.cse.ucsc.edu>) including human. Since DNA sequence is made up of only four letters, it is wasteful to spend a “character” for each position. The 2bit format uses only 2 “bits” to uniquely identify every nucleic acid and thus saves an enormous amount of memory. With smaller files, a specific locus can be accessed more quickly too. Fortunately, you will have access to Python code that can read these files so we do not have to worry more about the format. We will come back to this—for now knowing that sequences can be stored as text is sufficient.

## Gene expression

Another common file format in bioinformatics is for gene expression data: the GEO/SOFT format. It is designed to capture information produced by microarrays, intensity levels of many thousands of probes (corresponding to sections of DNA). A line in the main section of a file first contains a probe identifier, a gene name followed by (on the same line) a possibly large number of numeric values (intensities, possibly processed in some way), e.g.

ID_REF	IDENTIFIER	GSM31653	GSM31654	GSM31659	GSM31660	GSM31655
10000_at	YLR331C	9.300	6.500	7.600	3.700	4.800
10001_at	MID2	3.400	7.000	1.300	6.400	6.700
10002_i_at	RPS25B	6.100	4.500	3.100	7.300	3.200
...						

In the beginning of this file there is also information about what those values mean, and the following code designed to read such files disregard most of this sometimes-important information. The following function will read a GEO file and create a dictionary with an entry for each probe/gene name (key) and all numeric values.

```
def readGeoFile(filename, id_column = 0):
    """ Read a Gene Expression Omnibus SOFT file. """
    dataset = None
    fh = open(filename, "rU")
    manylines = fh.read()
    fh.close()
    data_rows = False # Are we reading data or metadata?
    name = 'Unknown'
    cnt_data = 0
    for line in manylines.splitlines():
        if line.startswith('^DATASET'):
            name = line.split('= ')[1]
            continue
        data_rows = line.startswith('!dataset_table_begin')
        data_rows = not line.startswith('!dataset_table_end')
        if len(line.strip()) == 0 or line.startswith('!') or \
            line.startswith('#') or line.startswith('^'):
            continue
        if data_rows:
            cnt_data += 1
            if (cnt_data == 1): # First line contains the headers
                headers = line.split('\t')
                dataset = {} # Create the data set
                continue
            ignore = (dataset == None) # ignore if not initialised
            id = line.split('\t')[id_column]
            values = []
            cnt_word = 0
            for word in line.split('\t'):
                cnt_word += 1
                if cnt_word <= (id_column + 1): # ignore gene names
                    continue
                if word == 'null':
                    ignore = True # ignore gene if a value is null
                    continue
                try:
                    values.append(float(word))
```

```

        except: # ignore values that are not "float"
            continue
        if not ignore:
            dataset[id] = values
    print 'Data set %s contains %d genes' % (name, len(dataset))
    return dataset

```

With this code in place (and see below for more information about putting things “in place”), you can read a GEO file that you perhaps downloaded from NCBI’s Gene Expression Omnibus (<http://www.ncbi.nlm.nih.gov/geo/>).

```

>>> ge = readGeoFile('GDS922.soft')
>>> ge['10001_at']
[3.4, 7.0, 1.3,      6.4, 6.7]

```

## Modules: Example classes Sequence and Alphabet

Previously we have represented biological sequences as text strings and we will keep doing that. However, biological sequences are made up of much fewer symbols than there are characters. When we analyse sequences we need to define all possible symbols that can occur, so we can find errors, count them and define “sequence patterns” etc. So we will define an Alphabet class. This will be used to define the Sequence class, which will have methods for counting and finding symbols. All of that will be made permanent by means of so-called modules. Then we can truly start doing serious work with sequences.

A module is a file with Python code. When you want to use this code (in the interpreter or another module) you import it. Let’s type the following into a file that you call “sequence.py” (you can use TextWrangler or any text editor to do this). Alternatively, all of this is available almost verbatim in a file called “guide.py”:

```

class Alphabet():
    """ A minimal class for alphabets """
    def __init__(self, symbolString):
        self.symbols = symbolString
    def __len__(self): # implements the "len" operator
        return len(self.symbols)
    def __contains__(self, sym): # implements the "in" operator
        return sym in self.symbols
    """ Below we declare alphabets that are going to be available when
    this module is imported """
    DNA_Alphabet = Alphabet('ACGT')
    RNA_Alphabet = Alphabet('ACGU')
    Protein_Alphabet = Alphabet('ACDEFGHIKLMNPQRSTVWY')

class Sequence():
    """ A biological sequence class. Stores the sequence itself,
    the alphabet and a name. """
    def __init__(self, sequence, alphabet, name):
        for sym in sequence:
            if not sym in alphabet: # error check: bail out
                raise RuntimeError('Invalid symbol: ' + sym)
        self.sequence = sequence
        self.alphabet = alphabet
        self.name = name

```

```

def __len__(self):          # the "len" operator
    return len(self.sequence)
def writeFasta(self):
    """ Write one sequence in FASTA format to a string and
        return it. """
    fasta = '>' + self.name + '\n'
    data = self.sequence
    nlines = (len(self.sequence) - 1) / 60 + 1
    for i in range(nlines):
        lineofseq = ''.join(data[i*60 : (i+1)*60]) + '\n'
        fasta += lineofseq
    return fasta
def __str__(self):
    return self.writeFasta()
def getcount(self, findme):
    """ Get the number of occurrences of specified symbol """
    cnt = 0
    for sym in self.sequence:
        if findme == sym:
            cnt = cnt + 1
    return cnt
def find(self, findme):
    """ Find the position of the symbol or sub-sequence """
    return self.sequence.find(findme)

```

These classes are not specific to DNA, RNA or proteins. They can be used for virtually any size alphabet or any length of sequence. The `__init__` function ensures that any alphabet or sequence is properly initialised before they are used: an alphabet is necessarily defined in terms of a list of symbols; a sequence always has a name, an alphabet and a sequence that is checked so that it does not have any symbols not defined by the alphabet. (Note we create three variables that correspond to these in the module.)

There are many so-called operators in Python, e.g. `'=='`, `'in'` and `'len(obj)'` that are defined over objects from classes. Above, we define the `len` operator for both the alphabet and sequence class by implementing the function `__len__`. The `'in'` operator that is used on the alphabet class in the sequence `__init__` function (if not `sym in alphabet`) is defined by implementing the `__contains__` function (see the alphabet class).

To see how we can use this module called "sequence", start up the interpreter and type:

```

>>> from sequence import *
>>> myseq = Sequence('AGCCCGGAT', DNA_Alphabet, 'MyDNA')
>>> myseq.getcount('G')
3
>>> myseq.find('CGG')
4
>>> Shabby_DNA = Alphabet('ACGTN')
>>> shabbyseq = Sequence('NNGCNNNGAT', Shabby_DNA, 'ShabbyDNA')

```

The first row will look for a file called `sequence.py` in your `PYTHONPATH` or current directory. Sometimes, programmers use

```

>>> import sequence

```



which does the same thing, but it forces us to always refer to the module explicitly with the *modulename.* prefix, e.g.:

```
>>> myseq = sequence.Sequence('AGCCAT', DNA_Alphabet, 'MyDNA')
```

Lastly, type the following and see how Python can provide you with help whilst you are interacting with defined classes:

```
>>> help(Sequence)
```

Note that if you did not create your own file but instead wish to use `guide.py`, you simply specify that module to use the Alphabet and Sequence classes, e.g.

```
>>> from guide import *
```

## Web services

Much of the power bioinformatics methods offer derives from access to massive data sets, e.g. find similar sequences in model organisms to transfer knowledge from one to the other or to establish evolutionary relationships, discover statistically significant sequence patterns, or genome-wide gene expression profiles in a cell type of interest etc. Most experimental data sets are made available freely via various web services—something that we are now going to explore.

We use EBI (European Bioinformatics Institute which now has a counterpart hosted at UQ, EMBL Australia) web services to retrieve data including specific sequences, and to perform searches in Uniprot, and via BLAST (a well-known sequence similarity search engine) against many available DNA, RNA and protein data sets. We can use these services to retrieve many other types of data (e.g. gene expression) as well. We're providing the following code as part of `guide.py`—note that we in each case need to import and use the `urllib` and `urllib2` Python libraries. In a separate code repository you will have access to functions and classes for using online BLAST and Clustal multiple alignment.

### Sequence data retrieval

EBI provides a host of sequence data that can be retrieved by sequence identifier. The following methods query a database that you nominate and return one or more Sequence instances.

```
import urllib, urllib2

def getSequence(entryId, dbName, alphabet):
    """ Retrieve a single entry from a database
    entryId: ID for entry e.g. 'P63166' or 'SUMO1_MOUSE'
    dbName: name of db e.g. 'uniprotkb', 'pdb' or 'refseqn' """
    url = 'http://www.ebi.ac.uk/Tools/dbfetch/dbfetch?style=raw&db=' + \
    dbName + '&format=fasta&id=' + entryId
    try:
        data = urllib2.urlopen(url).read()
        return readFastaString(data, alphabet)[0]
    except urllib2.HTTPError, ex:
        raise RuntimeError(ex.read())

def searchSequences(query, alphabet):
    """ Retrieve multiple entries matching query from UniProtKB
```

```

query: search term(s) """
url = 'http://www.uniprot.org/uniprot/?format=fasta&query=' + query
try:
    data = urllib2.urlopen(url).read()
    return readFastaString(data, alphabet)
except urllib2.HTTPError, ex:
    raise RuntimeError(ex.read())
return

```

If you start Python, you can type

```

>>> from guide import *
>>> seq = getSequence('Q712N7', 'uniprotkb', Protein_Alphabet)
>>> print seq
trIQ712N7IQ712N7_HUMAN: PHRLNSHLKLGFEVDIAEPVT
>>> seqs = searchSequences('organism:9606+AND+sumo+AND+ligase',\
    Protein_Alphabet)
>>> print len(seqs)
61
>>> print seqs[0]
spIP17544IATF7_HUMAN: MGDDRPFVCNAPGCGQRFTNEDH...

```

## ID mapping

Unfortunately the number of different identifiers almost equals the number of databases. Thus, mapping between different identifiers is a common exercise. The following code uses Uniprot's excellent mapping service.

```

def idmap(identifiers, frm='ACC', to='P_REFSEQ_AC'):
    """
    Map identifiers between databases (based on UniProt's service).
    identifiers: a list of identifiers (one or list of strings)
    frm: the abbreviation for the identifier FROM which to idmap
    to: the abbreviation for the identifier TO which to idmap
    Returns a dictionary with key (from) -> value (to) """
    url = 'http://www.uniprot.org/mapping/'
    # construct query by concatenating the list of identifiers
    if isinstance(identifiers, str):
        query = identifiers.strip()
    else: # assume it is a list of strings
        query = ''
        for id in identifiers:
            query = query + id.strip() + ' '
        query = query.strip() # remove trailing spaces
    params = {
        'from' : frm,
        'to' : to,
        'format' : 'tab',
        'query' : query
    }
    if len(query) > 0:
        request = urllib2.Request(url, urllib.urlencode(params))
        response = urllib2.urlopen(request).read()
        d = dict()
        for row in response.splitlines()[1:]:
            pair = row.split('\t')
            d[pair[0]] = pair[1]
        return d
    else:

```

```
return dict()
```

Here's an example for mapping from a Uniprot accession identifier to NCBI's Refseq protein identifier. (You can find a list of different types of valid identifiers at <http://www.uniprot.org/faq/28>.)

```
>>> id_map = idmap('P63166', frm='ACC', to='P_REFSEQ_AC')
>>> print id_map
{'P63166': 'NP_033486.1'}
```

## Gene Ontology

The GO is a systematic annotation of all genes, using a controlled dictionary (GO terms). For querying the Gene ontology, we rely again on EBI's Gene Ontology service. See <http://www.ebi.ac.uk/QuickGO/WebServices.html> for more technical information.

```
def getGODef(goterm):
    """
    Retrieve information about a GO term
    goterm: the identifier, e.g. 'GO:0002080'
    """
    # Construct URL
    url = 'http://www.ebi.ac.uk/QuickGO/GTerm?format=obo&id=' + goterm
    # Get the entry: fill in the fields specified below
    try:
        entry = {'id': None, 'name': None, 'def': None}
        data = urllib2.urlopen(url).read()
        for row in data.splitlines():
            index = row.find(':')
            if index > 0 and len(row[index:]) > 1:
                field = row[0:index].strip()
                value = row[index+1:].strip(' "') # remove spaces
                if field in entry.keys(): # check if we need field
                    if entry[field] == None: # check if assigned
                        entry[field] = value
        return entry
    except urllib2.HTTPError, ex:
        raise RuntimeError(ex.read())

def getGOTerms(genes, db='UniProtKB'):
    """
    Retrieve all GO terms for a given set of genes (or single gene).
    db: use specified database, e.g. 'UniProtKB', 'UniGene',
    or 'Ensembl'.
    The result is given as a map (key=gene name, value=list of unique
    terms) OR in the case of a single gene as a list of unique terms.
    """
    if type(genes) != list and type(genes) != set and type(genes) != tuple:
        genes = [genes] # if a single gene, we make a single item list
    map = dict()
    uri = 'http://www.ebi.ac.uk/QuickGO/GAnnotation?format=tsv&db='+db+'&protein='
    for gene in genes:
        terms = set() # empty result set
        url = uri + gene.strip() # Construct URL
        try: # Get the entry: fill in the fields specified below
            data = urllib2.urlopen(url).read()
            for row in data.splitlines()[1:]: # we ignore header row
```

```

        values = row.split('\t')
        if len(values) >= 7:
            terms.add(values[6]) # add term to result set
        map[gene] = list(terms) # make a list of the set
    except urllib2.HTTPError, ex:
        raise RuntimeError(ex.read())
if len(genes) == 1:
    return map[genes[0]]
else:
    return map

def getGenes(goterms, db='UniProtKB', taxo=None):
    """
    Retrieve all genes/proteins for a given set of GO terms
    (or single GO term).
    db: use specified database, e.g. 'UniProtKB', 'UniGene', or 'Ensembl'
    taxo: use specific taxonomic identifier, e.g. 9606 (human)
    The result is given as a map (key=gene name, value=list of unique
    terms) OR in the case of a single gene as a list of unique terms. """
    if type(goterms) != list and type(goterms) != set and type(goterms) != tuple:
        goterms = [goterms]
    map = dict()
    if taxo == None:
        uri = 'http://www.ebi.ac.uk/QuickGO/GAnnotation?format=tsv&db='+db+'&term='
    else:
        uri = 'http://www.ebi.ac.uk/QuickGO/GAnnotation?format=tsv&db='+db+'&tax='+\
            str(taxo)+'&term='
    for goterm in goterms:
        genes = set() # start with empty result set
        url = uri + goterm.strip() # Construct URL
        try: # Get the entry: fill in the fields specified below
            data = urllib2.urlopen(url).read()
            for row in data.splitlines()[1:]: # we ignore header
                values = row.split('\t')
                if len(values) >= 7:
                    genes.add(values[1]) # add gene name to result set
            map[goterm] = list(genes)
        except urllib2.HTTPError, ex:
            raise RuntimeError(ex.read())
    if len(goterms) == 1:
        return map[goterms[0]]
    else:
        return map

```

Let's try it:

```

>>> from guide import *
>>> print getG0Def('GO:0002080')['name']
acrosomal membrane
>>> print getG0Terms(['Q9SJN0', 'P63166'])
{'P63166': ['GO:0005737', 'GO:0016607', 'GO:0016605', 'GO:0031625'...
>>> print getGenes(['GO:0002080'], taxo=9606)
['P51636', 'Q2TNI1', 'Q8IXA5', 'C9JS54', 'Q6FG43', 'C9JR15'...

```

## Appendix 1: A quick tour of the BINF code

Over the last year or so we have developed a more powerful version of the functions mentioned in the previous sections of this document. This version is made available as a set of modules that are explained in the forthcoming sections. (This includes a module “sequence.py” that we started coding in an earlier section.) You should view these as a resource that you can use to solve bioinformatics problems in the course (and beyond) without getting too concerned with details and/or to learn about how standard bioinformatics methods can be implemented in Python.

### **sequence.py**

This module incorporates classes for `Sequence` (names and defines a sequence of symbols; computes various transformations and pairwise alignments); `Alignment` (defines a multiple sequence alignment; computes stats for use in substitution matrices); `SubstMatrix` (class to support alignment methods); `Regex` (defines patterns as regular expressions for textual pattern matching in sequences); and `PWM` (defines a weight matrix that can score any site in actual sequences). This module depends on two other modules, namely `symbol` and `prob` (explained below).

It also incorporates methods for loading and saving files relevant to the above (e.g. FASTA, ALN, substitution matrices) and methods for retrieving relevant data from web services (all of which are implemented in the `webservice` module explained below).

### **symbol.py**

Module `symbol` is for defining alphabets (of symbols), and for storing and operating on symbols and tuples (ordered or unordered). The module defines a class `Alphabet` that holds allowable characters, e.g. the alphabet for DNA is A, G, C and T. An alphabet is required to create instances of `Sequence` (see `sequence.py`).

We also use alphabets to define “tuple” tables where entries are keyed by combinations of symbols of an alphabet (see class `TupleStore`). This means that we can define probability distributions over short sequences (see `prob.py` below).

### **prob.py**

This module defines structures to hold probabilities (`prob` also depends on `symbol`). It defines classes and functions that are representing and processing basic probabilities. Uses and depends on `Alphabet` and `TupleStore` to define discrete random variables.

The `prob` module defines classes such as `Distrib` (for a discrete probability distribution, defined over single symbols from a specified `Alphabet`). Furthermore, it defines the class `Joint` to represent a joint probability. The joint probability is represented as a distribution over  $n$ -tuples where  $n$  is the number of variables. Variables can be for any `Alphabet`. The size of each alphabet determines the number of entries in the table (with probabilities that add up to 1.0). The `prob`

module also defines the class `NaiveBayes` that implements a classifier: a model defined over a class variable and conditional on a list of discrete feature variables.

### **webservice.py**

This module is collection of functions for accessing the EBI REST web services, including sequence retrieval, searching, gene ontology, BLAST and ClustalW. The class `EBI` takes precautions taken as to not send too many requests when performing BLAST and ClustalW queries.

See [http://www.ebi.ac.uk/Tools/webservices/tutorials/01\\_intro](http://www.ebi.ac.uk/Tools/webservices/tutorials/01_intro), [http://www.ebi.ac.uk/Tools/webservices/tutorials/02\\_rest](http://www.ebi.ac.uk/Tools/webservices/tutorials/02_rest) and [http://www.ebi.ac.uk/Tools/webservices/tutorials/06\\_programming/python/rest/urllib](http://www.ebi.ac.uk/Tools/webservices/tutorials/06_programming/python/rest/urllib)

The module allows access to the QuickGO Gene Ontology service (<http://www.ebi.ac.uk/QuickGO/WebServices.html>). Note that this service can be slow for queries involving a large number of entries, so exercise caution.

### **seqdata.py**

If you start the UCSC Genome Browser you will notice specific genomic loci can be accessed by specifying (obviously) organism/genome, but also “assembly”, “chromosome”, and “position”. Assemblies tend to change as technologies improve our ability to recover a genome sequence from “sequencing data”. Most are provided in a variety of formats for download, but most conveniently in the 2bit format as described above.

`seqdata.py` provides methods and classes for working with genome sequence data.

For instance,

- BED files
- 2bit genome sequence files

Annotations of genome sequence are often provided in separate files—known as BED files (Browser Extensible Data). If you click on “Tables” in the Genome Browser, you can download “tracks”, say all “UCSC Genes”. For example, a BED file could look like this:

chr1	55424	59692	uc009vjh.1	0	+	58953	59691	0	3	12,83,793,	0,327,3475,
chr1	58953	59871	uc001aal.1	0	+	58953	59871	0	1	918,	0,
...											

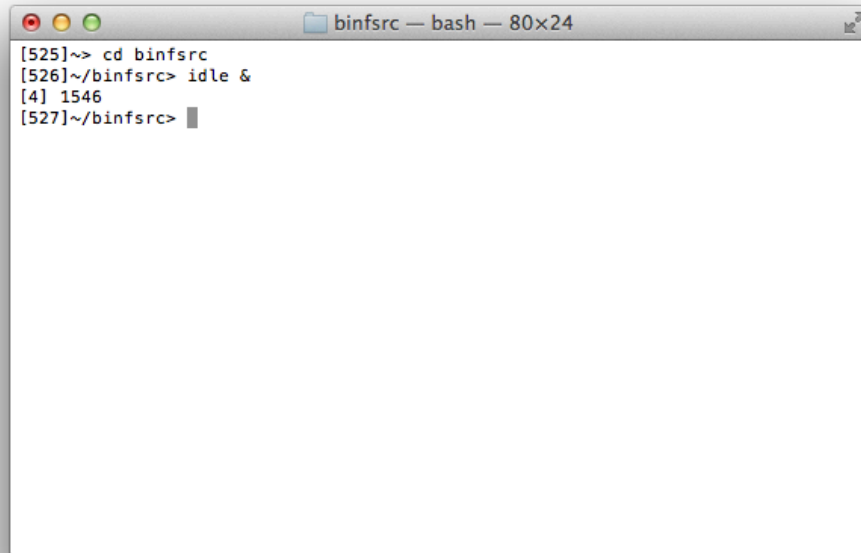
where the first three columns are the most important: chromosome identifier, start locus and end locus. Each row represents an entry, here position of gene (labelled “uc009vjh.1” and “uc001aal.1” in the example).

### **stats.py**

Module with methods for doing some statistics, including performing Fisher’s exact test and the Wilcoxon Ranksum test.

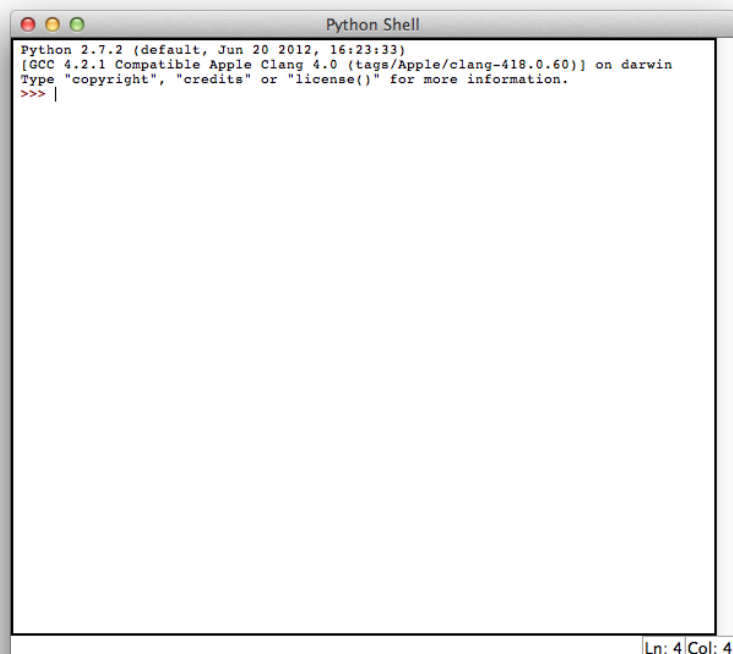
## Appendix 2: Running Python/IDLE on Mac OS X

1. Start a terminal
2. Start IDLE (should be in your search path if you have standard Python installation)



```
binfsrc — bash — 80x24
[525]~> cd binfsrc
[526]~/binfsrc> idle &
[4] 1546
[527]~/binfsrc> █
```

3. By default you enter into IDLE's "shell" where you can interact with Python



```
Python Shell
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Applet/clang-418.0.60)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> █
```

Ln: 4 | Col: 4

4. You can check where Python will look for “modules”, by importing “sys” – a systems “module”, and printing its “path” variable



```
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apples/clang-418.0.60)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> print sys.path
['/System/Library/Frameworks/Python.framework/Versions/2.7/bin', '/Users/mikael/
workspace/binf/src', '/Users/mikael/workspace/pbioalg', '/Users/mikael/binfsrc',
'/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7.zip', '/
System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7', '/System
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin', '/
System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac',
'/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-ma
c/lib-scriptpackages', '/System/Library/Frameworks/Python.framework/Versions/2.7
/Extras/lib/python', '/System/Library/Frameworks/Python.framework/Versions/2.7/l
ib/python2.7/lib-old', '/System/Library/Frameworks/Python.framework/Versions/2.7
lib/python2.7/lib-old', '/System/Library/Frameworks/Python.framework/Versions/2.
7/lib/python2.7/lib-dynload', '/System/Library/Frameworks/Python.framework/Versi
ons/2.7/Extras/lib/python/PyObjC', '/Library/Python/2.7/site-packages']
>>>
```

Ln: 6 Col: 159

5. If your own “modules” are in the search path, you can “import” them too



```
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apples/clang-418.0.60)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> print sys.path
['/System/Library/Frameworks/Python.framework/Versions/2.7/bin', '/Users/mikael/
workspace/binf/src', '/Users/mikael/workspace/pbioalg', '/Users/mikael/binfsrc',
'/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7.zip', '/
System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7', '/System
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin', '/
System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac',
'/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-ma
c/lib-scriptpackages', '/System/Library/Frameworks/Python.framework/Versions/2.7
/Extras/lib/python', '/System/Library/Frameworks/Python.framework/Versions/2.7/l
ib/python2.7/lib-old', '/System/Library/Frameworks/Python.framework/Versions/2.7
lib/python2.7/lib-old', '/System/Library/Frameworks/Python.framework/Versions/2.
7/lib/python2.7/lib-dynload', '/System/Library/Frameworks/Python.framework/Versi
ons/2.7/Extras/lib/python/PyObjC', '/Library/Python/2.7/site-packages']
>>> from guide import *
Sequence x: ACTGA is constructed from the symbols ACGT
( There are 2 occurrences of the symbol 'A' in ACTGA )
Sequence y: TACGA is constructed from the symbols ACGT
( The sub-sequence 'CG' starts at index 2 of TACGA )

Below is a substitution matrix for the alphabet ACGT
A 2
C -1 2
G -1 -1 2
T -1 -1 -1 2
A C G T

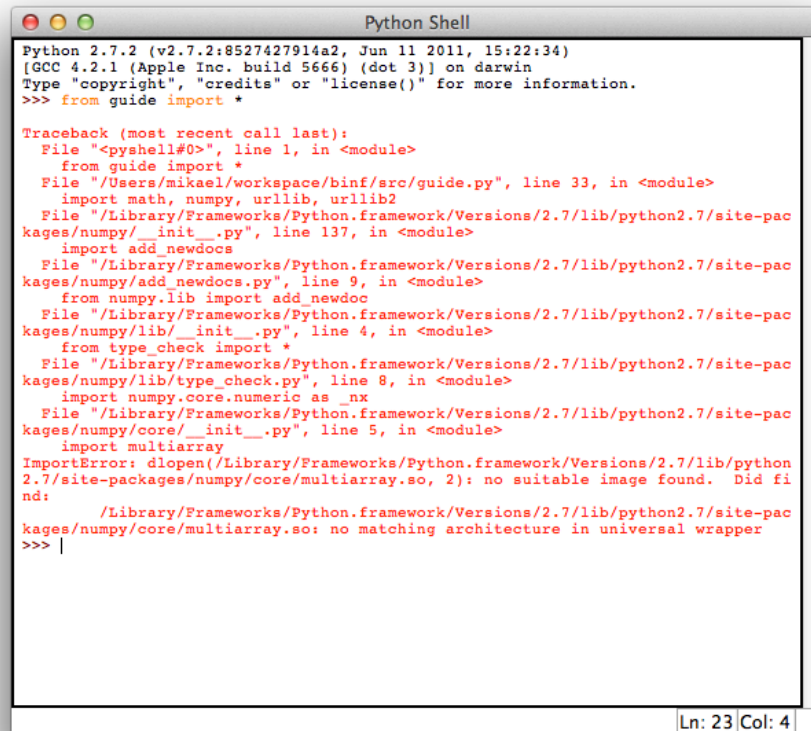
Below is the alignment between x and y
x -ACTGA
y TACGA

>>> |
```

Ln: 25 Col: 4



6. Chances are that something goes wrong—here is how it may look if your installation is incomplete

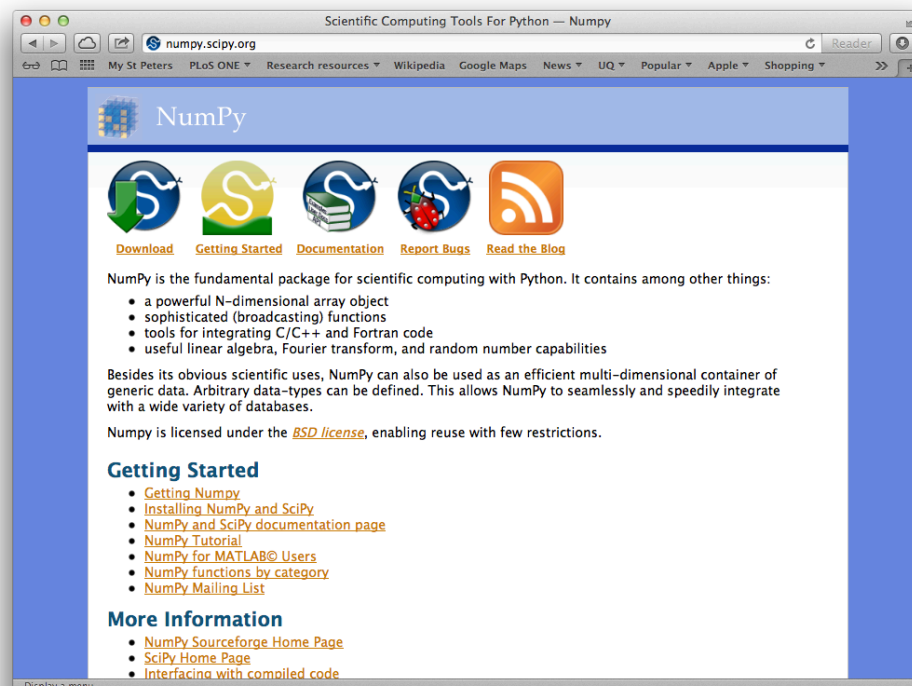


```
Python 2.7.2 (v2.7.2:8527427914a2, Jun 11 2011, 15:22:34)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> from guide import *

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    from guide import *
  File "/Users/mikael/workspace/binf/src/guide.py", line 33, in <module>
    import math, numpy, urllib, urllib2
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/__init__.py", line 137, in <module>
    import add_newdocs
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/add_newdocs.py", line 9, in <module>
    from numpy.lib import add_newdoc
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/lib/__init__.py", line 4, in <module>
    from type_check import *
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/lib/type_check.py", line 8, in <module>
    import numpy.core.numeric as _nx
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/core/__init__.py", line 5, in <module>
    import multiarray
ImportError: dlopen(/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/core/multiarray.so, 2): no suitable image found. Did find:
    /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy/core/multiarray.so: no matching architecture in universal wrapper
>>> |
```

Ln: 23 Col: 4

7. The above problem could be rectified by installing the numerical python library (numpy/scipy) to go with your Python installation



8. Another potential issue is less problematic: what if your Python search path does not include the directory where you put Python code?



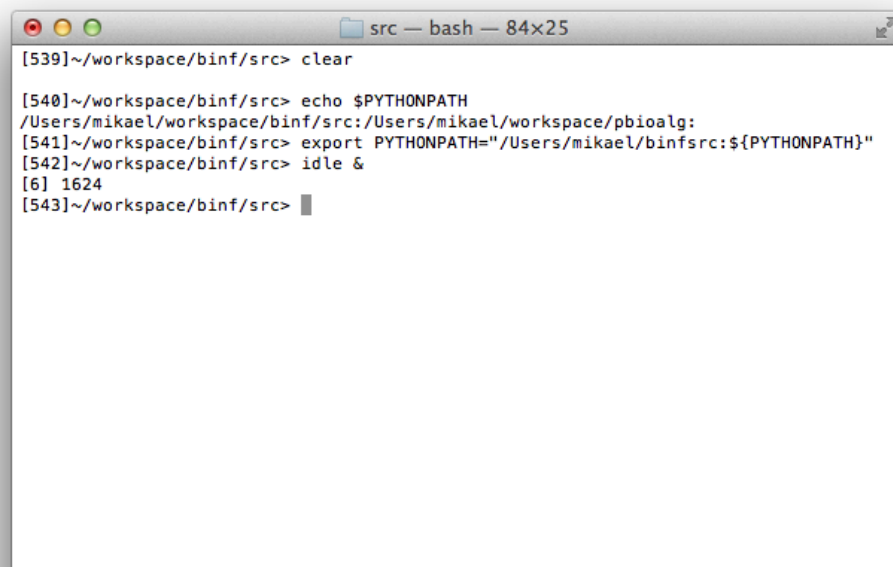
A screenshot of a 'Python Shell' window. The window title is 'Python Shell'. The text inside shows the Python version (2.7.2) and compiler (GCC 4.2.1). The user enters the command `>>> from guide import *`. This results in a `Traceback` error: `File "<pyshell#0>", line 1, in <module> from guide import * ImportError: No module named guide`. The status bar at the bottom right indicates 'Ln: 10 Col: 4'.

```
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Applet/clang-418.0.60)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> from guide import *

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    from guide import *
ImportError: No module named guide
>>>
```

Ln: 10 Col: 4

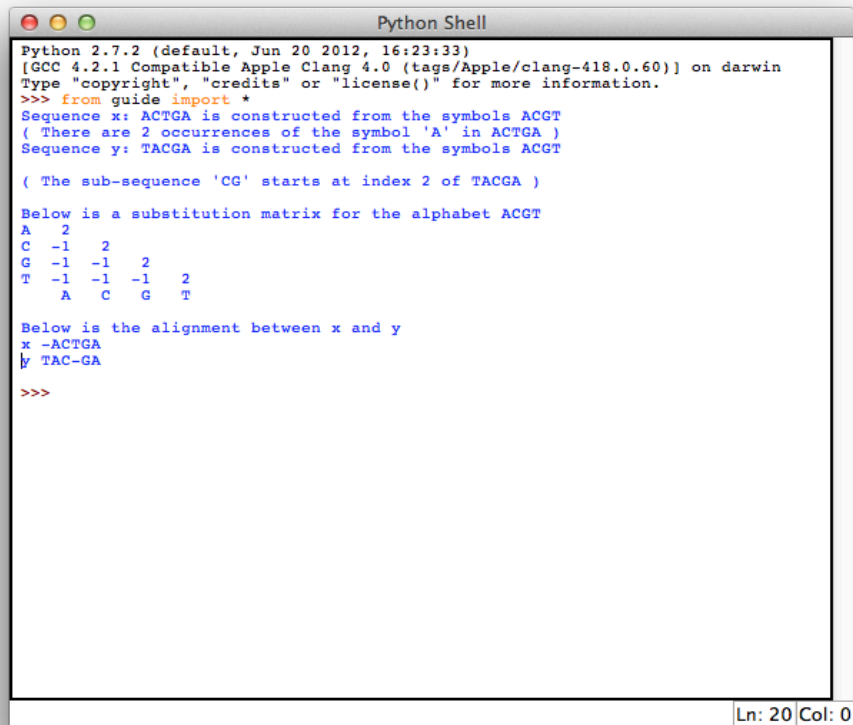
9. Quit IDLE and fix the OS system variable, like so



A screenshot of a terminal window titled 'src — bash — 84x25'. The user is in the directory `~/workspace/binf/src`. The commands and their outputs are as follows:

- `[539]~/workspace/binf/src> clear`
- `[540]~/workspace/binf/src> echo $PYTHONPATH`  
Output: `/Users/mikael/workspace/binf/src:/Users/mikael/workspace/pbioalg:`
- `[541]~/workspace/binf/src> export PYTHONPATH="/Users/mikael/binfsrc:${PYTHONPATH}"`
- `[542]~/workspace/binf/src> idle &`
- `[6] 1624`
- `[543]~/workspace/binf/src>`

10. Restart and you should be fine



```
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Appletclang-418.0.60)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> from guide import *
Sequence x: ACTGA is constructed from the symbols ACGT
( There are 2 occurrences of the symbol 'A' in ACTGA )
Sequence y: TACGA is constructed from the symbols ACGT

( The sub-sequence 'CG' starts at index 2 of TACGA )

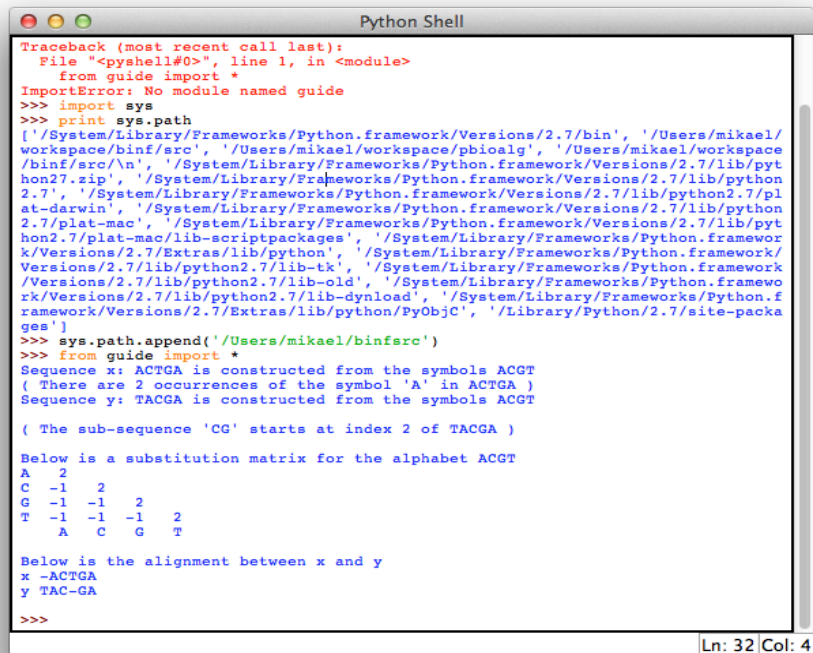
Below is a substitution matrix for the alphabet ACGT
A   2
C  -1   2
G  -1  -1   2
T  -1  -1  -1   2
   A   C   G   T

Below is the alignment between x and y
x -ACTGA
y TAC-GA

>>>
```

Ln: 20 Col: 0

11. Alternatively, stay in IDLE and manually append your directory to the internally kept search path, like so



```
Python Shell
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    from guide import *
ImportError: No module named guide
>>> import sys
>>> print sys.path
['/System/Library/Frameworks/Python.framework/Versions/2.7/bin', '/Users/mikael/
workspace/binf/src', '/Users/mikael/workspace/pbioalg', '/Users/mikael/workspac
e/binf/src', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/pyt
hon27.zip', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/pytho
n2.7', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/pl
at-darwin', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/pytho
n2.7/plat-mac', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/pyt
hon2.7/plat-mac/lib-scriptpackages', '/System/Library/Frameworks/Python.framewo
rk/Versions/2.7/Extras/lib/python', '/System/Library/Frameworks/Python.framework
/Versions/2.7/lib/python2.7/lib-tk', '/System/Library/Frameworks/Python.framewo
rk/Versions/2.7/lib/python2.7/lib-old', '/System/Library/Frameworks/Python.frame
rk/Versions/2.7/lib/python2.7/lib-dynload', '/System/Library/Frameworks/Python.f
ramework/Versions/2.7/Extras/lib/python/PyObjC', '/Library/Python/2.7/site-packa
ges']
>>> sys.path.append('/Users/mikael/binfsrc')
>>> from guide import *
Sequence x: ACTGA is constructed from the symbols ACGT
( There are 2 occurrences of the symbol 'A' in ACTGA )
Sequence y: TACGA is constructed from the symbols ACGT

( The sub-sequence 'CG' starts at index 2 of TACGA )

Below is a substitution matrix for the alphabet ACGT
A   2
C  -1   2
G  -1  -1   2
T  -1  -1  -1   2
   A   C   G   T

Below is the alignment between x and y
x -ACTGA
y TAC-GA

>>>
```

Ln: 32 Col: 4