1 Prologue

Contents

Python programming for biology

One of the main aims of this book is to empower the average researcher in the life sciences, who may have a pertinent scientific question that can be readily answered by computational techniques, but who doesn't have much, if any, experience with programming. For many in this position, the task of writing a program in a computer language is a bottleneck, if not an impassable barrier. Often, the task is daunting and seems to require a significant investment of time. The task is also subject to the barriers presented by a vocabulary filled with jargon and a seemingly steep learning curve for those people who were not trained in computing or have no inclination to become computer specialists. With this in mind for the novice programmer, one ought to start with the language that is the easiest to get to grips with, and at the time of writing we believe that that language is Python. This is not to say that we have made a compromise by choosing a language that is easy to learn but which is not powerful or fully featured. Python is certainly a very rich and capable way of programming, even for very large projects; otherwise we authors wouldn't be using it for our own scientific work.

A second main aim of this book is to use Python as a means to illustrate some of what is going on within biological computing. We hope our explanations will show you the scientific context of why something is done with computers, even if you are a newcomer to biology or medical sciences. Even where a popular biological program is not written in Python, or if you are a programmer who has good reason for using another language, we can still use Python as a way of illustrating the major principles of programming for biology. We feel that many of the most useful biological programs are based on combinations of simple principles that almost anyone can understand. By trying to separate the core concepts from the obfuscation and special cases, we aim to provide an overview of techniques and strategies that you can use as a resource in your own research. Virtually all of the examples in this book are working code that can be run and are based on real problems or programs within biological computing. The examples can then be adapted, altered and combined to enable you to program whatever you need.

We wish to make clear that this book intends to show you what sort of things can be done and how to begin. It does not intend to offer a deep and detailed analysis of specific biological and computational problems. This is not a typical scientific book, given that we don't always go for the most detailed or up-to-date examples. Given the choice, we aim to give a broad-based understanding to newcomers and avoid what some may consider pedantry.

No doubt some people will think our approach somewhat too simplistic, but if you know enough to know the difference then we don't recommend looking to this book for those kinds of answers. Likewise, there is only room for so many examples and we cannot cover all of the scientific methods (including Python software libraries) that we would want to. Hopefully though, we give the reader enough pointers to make a good start.

Choosing Python

It is perhaps important to include a short justification to say why we have written this book for the Python programming language; after all, we can choose from several alternative languages. Certainly Python is the language that we the authors write in on a daily basis, but this familiarity was actually born out of a conscious decision to use Python for a large biological programming project after having tried and considered a number of popular alternatives. Aside from Python, the languages that we have commonly come across in today's biological community include: C, C++, FORTRAN, Java, Matlab, Perl, R and Ruby. Specific comparison with some of these languages will be made at various points in the book, but there are some characteristics of Python that we enjoy, which we feel would not be available to the same level or in the same combination in any other language.

We like the clear and consistent layout that directs the programmer away from obfuscated program code and towards an elegantly readable solution; this becomes especially important when trying to work out what someone else's program does, or even what your own material does several years later. We like the way that Python has object orientation at its heart, so you can use this powerful way to organise your data while still having the easy look and feel of Python. This also means that by learning the language basics you automatically become familiar with the very useful object-oriented approach. We like that Python generally requires fewer lines of program code than other languages to do the equivalent job, and that it often seems so much less tedious to write.

It is important to make it clear that we would not currently use Python for every programming task in the life sciences. Python is not a perfect language. As it stands currently for some specialised tasks, particularly those that require fast mathematical calculations which are not supported by the numeric Python modules, we actively promote working with a Python extension such as Cython, or some faster alternative language. However, we heartily recommend that Python be used to administer the bookkeeping while the faster alternative provides extra modules that act as a fast calculation engine. To this end, in Chapter 27 we will show you how you can seamlessly mesh the Python language with Cython and also with the compiled language C, to give all the benefits of Python and very fast calculations.

Python's history and versions

The Python[1] programming language was the creation of Guido van Rossum. It is because of his innovation and continuing support that Python is popular and continues to grow. The Python programming community has afforded Guido the honour of the title 'benevolent

[1] The name itself derives from Monty Python, which is why you'll find the occasional honorary reference to 'spam', 'dead parrot' etc. when arbitrary examples are given.

dictator for life'. What this means is that despite the fact that many aspects of Python are developed by a large community, Guido has the ultimate say in what goes into Python. Although not bound in any legality, everyone abides by Guido's decisions, even if at times some people are surprised by what he decides. We believe that this situation has largely benefited Python by ensuring that the philosophy remains unsullied. Seemingly often, a committee decision has the tendency to try to appease all views and can become tediously slow with indecision; too timid to make any bold, yet improving moves. The Python programming community has a large role in criticising Python and guiding its future development, but when a decision needs to be made, it is one that everyone accepts. Certainly there could be a big disagreement in the future, but so far the benevolent dictator's decisions have always taken the community with him.

There have been several, and in our opinion improving, versions of the Python programming language. All versions before Python 3 share a very high degree of backward-compatibility, so that code written for version 1.5 will still (mostly) work with say version 2.7 with few problems. Python 3 is not as compatible with older versions, but this seems a reasonable price to be paid to keep things moving forward and eradicate some of the undesired legacy that earlier versions have built up. Rest assured though, version 3 remains similar enough in look and feel to the older Pythons, even if it is not exactly the same, and the examples in this book work with both Python 2 and Python 3 except where specifically noted. Also, included with the release of Python 3 is a conversion program '2to3' which will attempt to automatically change the relevant parts of a version 2 program so that it works with version 3. This will not be able to deal with every situation, but it will handle the vast majority and save considerable effort.

For this book we will assume that you are using Python version 2.6 or 2.7 or 3. Some bits, however, that use some newer features will not work with versions prior to 2.6 without alteration. We feel that it is better to use the best available version, rather than write in a deliberately archaic manner, which would detract from clarity.

Bioinformatics

The field of bioinformatics has emerged as we have discovered, through experimentation, large amounts of DNA and protein sequence information. In its most conservative sense bioinformatics is the discipline of extracting scientific information by the study of these biological sequences, which, because of the large amount of data, must be analysed by computer. Initially this encompassed what most biological computing was about, but we contend that this was simply where biomolecular computing began and that it has far to go. The informatics of biological systems these days includes the study of molecular structures, including their dynamics and interactions, enzymatic activity, medical and pharmacological statistics, metabolic profiles, system-wide modelling and the organisation of experimental procedures, to name only a subset. It is within this wider context that this book is placed.

At present the programming language that is historically most famous for being used with bioinformatics is probably Perl, which is notable for its ability to manipulate sequences, particularly when stored as letters within formatted text. It also has a library of modules available to perform many common bioinformatics tasks, collectively named BioPerl. In this arena Python can do everything that Perl can. There is a Python equivalent of BioPerl,

unsurprisingly named BioPython, and at this time the uptake of Python within the bioinformatics community is growing, which is not surprising, given our belief that it is an easier but more powerful language to work with. It is important to note that although some of the BioPython modules will certainly be discussed in the course of this book (and we would generally advise using tested, existing code wherever possible to make your programs easier to write and understand) the explanations and examples will be more to do with understanding what is going on underneath. We aim to avoid this book simply becoming a brochure for existing programs where you don't have to know the inner workings.

Computer platforms and installations

Python is available for every commonly used computer operating system including versions of Microsoft Windows, Mac OS X, Linux and UNIX. With Windows you will generally have to download and install Python, as it is not included as standard. On most new Mac OS X, Linux and UNIX systems Python is included as standard (indeed some parts of Linux operating systems are themselves written with Python), although you should check to see which version of Python you have: typing '`python`' at a command line reveals the version. For a list of website locations where you can download Python for various platforms see the reference section at the end of this book or the Cambridge University Press site: http://www.cambridge.org/pythonforbiology.

Precisely because Python is available for and can be run on many different computer platforms, any programs you write will generally be able to be run on all computer systems. However, there are a few important caveats you should be aware of. Although Python as a language is interpreted in the same way on every computer system, when it comes to interacting with the operating system (Windows, Mac OS X, Linux . . .), things can work differently on different computers. This is a problem that all cross-platform computing languages face. You will probably come across this in your Python programs when dealing with files and the directories that contain them. Although each operating system will have its own nuances, once you are aware of the differences it is a relatively simple job to ensure that your programs work just as well under any common operating system, and we will cover details of this as required in the subsequent chapters.

2 A beginners' guide

Contents

Programming principles

The Python language can be viewed as a formalised system of understanding instructions (represented by letters, numbers and other funny characters) and acting upon those directions. Quite naturally, you have to put something in to get something out, and what you are going to be passing to Python is a series of commands. Python is itself a computer program, which is designed to interpret commands that are written in the Python language, and then act by executing what these instructions direct. A programmer will sometimes refer to such commands collectively as 'code'.

Interpreting commands

So, to our first practical point; to get the Python interpreter to do something we will give it some commands in the form of a specially created piece of text. It is possible to give Python a series of commands one at a time, as we slowly type something into our computer. However, while giving Python instructions line by line is useful if you want to test out something small, like the examples in this chapter, for the most part this method of issuing commands is impractical. What we usually do instead is create all of the lines of text representing all the instructions, written as commands in the Python language, and store the whole lot in a file. We can then activate the Python interpreter program so that it reads all of the text from the file and acts on all of the commands issued within. A series of commands

that we store together in such a way, and which do a specific job, can be considered as a computer program.[1] If you would like to try any of the examples given in the book the next chapter will tell you how to actually get started. The initial intention, however, is mostly to give you a flavour of Python and introduce a few key principles.

```
mass = 5.9736
volume = 1.08321
density = mass/volume
print(density)
```

An example of a very simple, four-line Python program that performs a calculation and displays the result.

Reusable functionality

When writing programs in the Python language, which the Python interpreter can then use, we are not restricted to reading commands from only one file. It is a very common practice to have a program distributed over a number of different files. This helps to organise writing of the program, as you can put different specialised parts of your instructions into different files that you can develop separately, without having to wade through large amounts of text. Also, and perhaps most importantly, having Python commands in multiple files enables different programs to share a set of commands. With shared files, the distinction between which commands belong to one program and which belong to another is mostly meaningless. As such, we typically refer to such a shared file as a *module*.

In Python you will use modules on a regular basis. And, as you might have already guessed, the idea is to have modules containing a series of commands which perform a function that would be useful for several programs, perhaps in quite different situations. For example, you could write a module which contains the commands to do a statistical analysis on some numeric data. This would be useful to any program that needs to run that kind of analysis, as hopefully we have written the statistics module in such a way that the precise amount and source of the numeric data that we send to the module is irrelevant. Whenever we use a module we are avoiding having to write new Python commands, and are hopefully using something that has been tried and tested and is known to work.

```
from Alignments import sequenceAlign
sequence1 = 'GATTACAGC'
sequence2 = 'GTATTAAT'
print(sequenceAlign(sequence1, sequence2))
```

A Python example where general functionality, to align two sequences of letters, is imported from a module called `Alignments`*, which was defined elsewhere.*

When working with Python there is already a long list of pre-made modules that you can use. For example, there are modules to perform common mathematical operations, to interact with

[1] Not 'programme', even in the UK.

the operating system and to search for patterns of symbols within text. These are all generally very useful, and as such they are included as standard whenever you have Python installed. You will still have to load, or *import*, these modules into a program to use them, but in essence you can think of these modules as a convenient way of extending the vocabulary of the Python language when you need to. By the same token, you don't have to load any modules that are not going to be useful, which might slow things down or use unnecessary computer memory.

Types of data

Before going on to give a more detailed tutorial we will first describe a little about the construction and makeup of commands written in the Python language. Writing the command code for a program involves thinking about items of data. There can be many different kinds of data, from different origins, that we would wish to manipulate with a computer. Typically we will represent the smallest units of this information as numbers or text. We can organise such numbers and text into structured arrangements, for example, to create a list of data, and we can then manipulate this entire larger container, with all of its underlying elements, as a single unit. For example, given a list containing numbers you could extract the first number from the list, or maybe get the list in reverse order.

```
numbers = [6, 0, 2, 2, 1, 4, 1, 5]
numbers.reverse()

print(numbers)
```

Defining a list of numbers as a single entity and then reversing its order, before printing the result to the screen.

In Python, as in many languages, there are some standard types of data-containing structures that form the basis of most programs, and which are very easy to create and fill with information. But you are not limited to these standard data structures; you can create your own data organisation. For example, you could create a data structure called a `Person`, which can store the name, sex, height and age of real people. In a program, just as you could get the first element of data stored in a list, so too could you extract the number that represents the age of a `Person` data structure. Going further, you could create many `Person` data structures and organise them further by placing them into lists. A data structure can appear inside the organisation of many other data structures, so a single `Person` could appear in several different lists (for example, organised by age, sex or whatever) or a `Person` could contain references to other `Person` data structures to indicate the relationships between parents and children.

Python objects

This is where we can introduce the concept of an *object*. The `Person` data structure described above would commonly be referred to as a `Person` object. Indeed, all of the organised data structures in Python, including the simple inbuilt ones, are referred to as Python objects. So numbers, text and lists are all kinds of objects. Not every programming language formalises things in this way, but it will start to feel natural once you are used to Python, and means that the form of the programming language is the same whatever type of object is being manipulated.

```
x = 3
y = 7
print(x + y)
print(x.__add__(y))
```

An example which shows the underlying object-oriented nature of numbers in Python: the last two lines do the same thing. Although we would normally write additions in a conventional way with a plus symbol, we are actually invoking the `__add__` *operation which all Python numbers possess.*

An important concept when dealing with objects is inheritance. That is to say that we can make a new type of data structure by basing it on an existing one. Indeed, every object in Python, except the simplest data structure of them all (the base object), inherits its organisation from another object. Accordingly, you could take a `Person` object and use its specification to create a `Scientist` object. This would immediately give the `Scientist` object the same data organisation of a `Person` object, with its age, sex and height data, but we can go on to modify the `Scientist` object to also store different information, like a list of publications or current work institution. This can also be done for the built-in objects, so you could have your own version of a Python list that is only allowed to contain odd numbers, if you really, really wanted.

So far we have discussed the manipulation of data by a Python program in fairly loose terms, so it is about time to more properly introduce you to a few of the concepts that you will commonly use in Python programs. The examples that we give use operations and types of data that are built into the language as standard, i.e. that the Python interpreter will know how to handle without you having to add any special information.

Variables

As will already be apparent from the above Python snippets, when you refer to some data in your program you will often be assigning it with a name, like '`x`' or '`sequence1`' or '`dnaList`'. Such names are commonly referred to as *variables*. They provide you with an identifiable label that you can use to track an individual item of data amongst many others within your program. The jargon term 'variable' is quite apt because you often want to keep the same name label but vary the value of the data it refers to. For example, you can write a program that calculates `x+2` and `x-2`; where `x` can be set to any numeric value and both operations are performed on that same named item, whatever it may be. This concept is similar to algebra, where you can describe formulae, like $y = x^2 + 3$, without specifying what `x` or `y` actually are, and then use the formula on different values of `x` in order to compute `y`.

Note that in Python if you set the variable with the name '`x`' to take a numeric value you can still set it to be some other type of data later on in the program, so initially it may be a number, but later be some text. Bearing this in mind, you must be careful that you only perform operations on the '`x`' data that are valid for that type of data. Staying with the idea of data items having a particular *data type*, we next go through the basic types available in Python.

Basic data types

Numbers

There are two common types of numeric data in Python. These are *integers*, the whole numbers, and *floating point* numbers, numbers with decimal points.

Integers

Integers are whole numbers and can be positive, negative or zero in value. You would typically use integers to count things that only come as a whole, like the size of a list or number of people. You can naturally perform mathematical operations with integers, also in combination with other types of number object, but in Python 2 if you perform some mathematical operations with only integers the result is an integer too. While this makes sense for addition and multiplication, division will give you the perhaps surprising result of a whole number, rounding the answer (towards negative infinity to be precise). The advantage of integer operations is that they are quick and always precise; non-integer representation can give rise to small errors which can sometimes have serious consequences.

In Python 2 there are actually two types of integers, normal integers and long integers, although you usually don't have to pay much attention to this fact. The long integer variety is used when the number is so big[2] that it must be stored in a different way, as it takes up more memory slots to store the digits. Accordingly, you might see the 18-digit number 123,456,789,123,456,789 represented in Python (before version 3) as `123456789123456789L`, i.e. with an extra `'L'` at the end giving a hint that it is the long variety. But otherwise you can simply treat it as a number and do all the usual operations with it. In Python 3 this distinction disappears and every integer is a long integer.

Floating point numbers

Floating point numbers, often simply referred to as *floats*, are numbers expressed in the decimal system, i.e. `2.1`, `999.998`, `0.000004` or whatever. The value `2.0` would also be interpreted as a floating point number, but the value `2`, without the decimal point, will not; it will be interpreted as an integer. Floating point numbers can also carry a suffix that states which power of ten they operate at. So, for example, you can express four point six million as 4.6×10^6, which in Python would be written as `4.6e6` (or as `46e5` or as `0.46e7`) and similarly one hundredth would be `1.0e-2`. A potential pitfall with floating point numbers is that they are of limited precision. Of course you would not expect to be able to express some fractions like ⅓ exactly, but there can otherwise be some surprises when you do certain calculations. For example, `0.1` plus `0.2` may sometimes give you something like `0.30000000000000004`, because of the way that the innards of computers work. The difference between this number and the desired value of `0.3` is what would be referred to as a *floating point error*. Often there is sufficient accuracy that a very small error doesn't matter, but sometimes it does matter and you should be aware of this issue. Common situations where the floating point errors could matter include: when you are repeatedly updating a value and the error grows, when you are interested in the small difference that results when subtracting two larger numbers and when

[2] Typically the long integers start at 2^{31} or 2^{63} depending on whether the system is 32 bit or 64 bit.

two values ought to be equal but they aren't exactly, e.g. after division you test for `1.0` but don't get the expected exact value.

Text strings

Strings are stretches of alphanumeric characters like `"abc"` or `'Hello world'`, in other words they represent text. In Python strings are indicated inside of single or double quotation marks, so that their text data can be distinguished from other data types and from the commands of the program. Thus if in Python we issue the command `print("lumberjack")` we know that `"lumberjack"` is the string data and everything else is Python command. Similarly, quotation marks will also distinguish between real numbers and text that happens to be readable as a number. For example, `1.71` is a floating point number but `"1.71"` is a piece of text containing four characters. You cannot do mathematics with the text string `"1.71"`, although it is possible to convert it to a number object with the value `1.71`.

String objects might contain elements that cannot be represented by the printable characters found on a keyboard, but which are nonetheless part of a piece of text. A good example of this is the way that you can split text over several lines. When you type into your computer you may use the *Return* key to do this. In a Python string you would use the special sequence `"\n"` to do this:[3] Python uses a combination of characters to provide the special meaning. For example, `"Dead Parrot"` naturally goes on one line, but `"Dead\nParrot"` goes on two, as if you had pressed *Return* between the two words.

Another concept that deserves some explanation is the empty string, written simply as `""`, with no visible characters between quotes. You can think of this in the same way as an empty list; as a data structure that is capable of containing a sequence, but which happens to contain nothing. The empty string is useful in situations where you must have a string object present but don't want to display any characters.

Text strings are made up of individual characters in a specific order, and in some ways you can think of them as being like lists. Thus, for example, you can query what the first character of a string is, or determine how long it is. In Python, however, you cannot modify strings once they are defined; if you want to make a change you have to recreate them in their entirety. This might seem stifling at first glance, but it rarely is in practice. The benefit of this system is that you can use strings to access items in a Python dictionary (which is a handy way to store data that we discuss below); if strings were internally alterable this would not be possible in Python. Python can readily perform operations to replace an existing string with a modified version. For example, if you wanted to convert some data that is initially stored as `"Dead Parrot"` into the text `"Ex-Parrot"` you could redefine the data as the string `"Ex-"` joined onto the last six characters of the original text. If at any point it really is painful to redefine a string entirely, a common trick is to convert the text into a list of separate characters (see list data type below) that you can manipulate internally, before converting the list of characters back into text.

Special objects

Booleans

The two Boolean objects are `True` and `False`, and they mean much what you might expect. Many objects can be examined to test whether they are logically false, like an

[3] To actually write the two characters `"\n"` without it being interpreted as a new line you would use `"\\n"`.