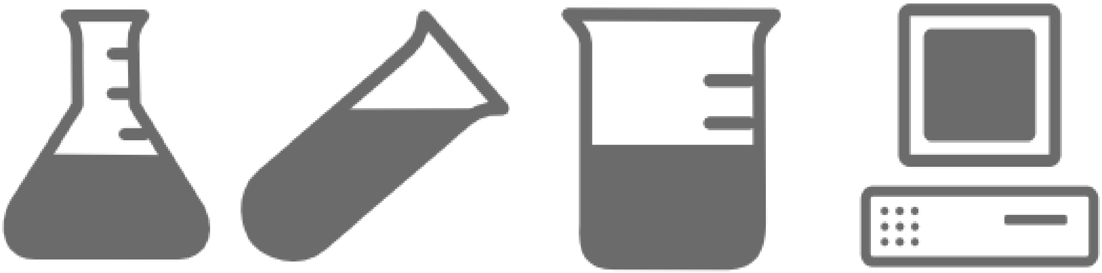
Python for Biologists



A programming course for complete beginners

by Dr. Martin Jones

http://pythonforbiologists.com

For more information, visit http://pythonforbiologists.com

Set in PT Serif and `Source Code Pro`

About the author

Martin started his programming career by learning Perl during the course of his PhD in evolutionary biology, and started teaching other people to program soon after. Since then he has taught introductory programming to hundreds of biologists, from undergraduates to PIs, and has maintained a philosophy that programming courses must be friendly, approachable, and practical.

Martin has taught introductory programming as part of the Bioinformatics MSc course at Edinburgh University for the past five years, and is currently Lecturer in Bioinformatics.

Preface

Welcome to Python for Biologists.

Before you read any further, make sure that this is the most recent version of the book. Python for Biologists is being continually updated and improved to take into account corrections, amendments and changes to Python itself, so it's important that you are reading the most up-to-date version.

This file is revision number **189**. The number of the most recent revision can always be found at:

http://pythonforbiologists.com/index.php/version/

If the revision number listed at the URL is higher than the one in bold, then this is an out-of-date copy, and you need to download the latest version from

http://pythonforbiologists.com

You'll notice from the copyright page that the contents of this book are licensed under a Creative Commons Attribution ShareAlike license. This means that you're free to do what you like with it – copy it, email it to your friends, wallpaper your lab with it – as long as you keep the attribution. You can also modify it, as long as you license your modification under the same terms. The only thing that the license doesn't allow is commercial use – if you'd like to use the contents of this course for commercial purposes, get in touch with me at

martin@pythonforbiologists.com

Happy programming!

Table of Contents

1: Introduction and environment

Why have a programming book for biologists?

If you're reading this book, then you probably don't need to be convinced that programming is becoming an increasingly essential part of the tool kit for biologists of all types. You might, however, need to be convinced that a book like this one, developed especially for biologists, can do a better job of teaching you to program than a general-purpose introductory programming book. Here are a few of the reason why I think that is the case.

A biology-specific programming book allows us to use examples and exercises that use biological problems. This serves two important purposes: firstly, it provides motivation and demonstrates the types of problems that programming can help to solve. Experience has shown that beginners make much better progress when they are motivated by the thought of how the programs they write will make their life easier! Secondly, by using biological examples, the code and exercises throughout the book can form a library of useful code snippets, which we can refer back to when we want to solve real-life problems. In biology, as in all fields of programming, the same problems tend to recur time and time again, so it's very useful to have this collection of examples to act as a reference – something that's not possible with a general-purpose programming book.

A biology-specific programming book can also concentrate on the features of the language that are most useful to biologists. A language like Python has many features and in the course of learning it we inevitably have to concentrate on some and miss others out. The set of features which are important to us in biology are slightly different to those which are most useful for general-purpose programming – for example, we are much more interested in manipulating text (including things like DNA and protein sequences) than the average programmer. Also, there are several features of Python that would not normally be discussed in an introductory programming book, but which are very useful to biologists (for example, regular expressions and subprocesses). Having a biology-specific textbook allows us to include these features, along with explanations of why they are particularly useful to us.

A related point is that a textbook written just for biologists allows us to introduce features in a way that allows us to start writing useful programs right away. We can do this by taking into account the sorts of problems that repeatedly crop up in biology, and prioritising the features that are best at solving them. This book has been designed so that you should be able to start writing small but useful programs using only the tools in the first couple of chapters.

Why Python?

Let me start this section with the following statement: programming languages are overrated. What I mean by that is that people who are new to programming tend to worry far too much about what language to learn. The choice of programming language does matter, of course, but it matters far less than people think it does. To put it another ways, choosing the "wrong" programming language is very unlikely to mean the difference between failure and success when learning. Other factors (motivation, having time to devote to learning, helpful colleagues) are far more important, yet receive less attention.

The reason that people place so much weight on the *"what language should I learn?"* question is that it's a big, obvious question, and it's not difficult to find people who will give you strong opinions on the subject. It's also the first big question that beginners have to answer once they've decided to learn programming, so it assumes a great deal of importance in their minds.

There are three main reasons why choice of programming language is not as important as most people think it is. Firstly, nearly everybody who spends any significant amount of time programming as part of their job will eventually end up using multiple languages. Partly this is just down to the simple constraints of various languages – if you want to write a web application you'll probably do it in Javascript, if you want to write a graphical user interface you'll probably use something like Java, and if you want to write low-level algorithms you'll probably use C.

Secondly, learning a first programming language gets you 90% of the way towards learning a second, third, and fourth one. Learning to think like a programmer in the way that you break down complex tasks into simple ones is a skill that cuts across all languages – so if you spend a few months learning Python and then discover that you really need to write in C, your time won't have been wasted as you'll be able to pick it up much quicker.

Thirdly, the kinds of problems that we want to solve in biology are generally amenable to being solved in any language, even though different programming languages are good at different things. In other words, as a beginner, your choice of language is vanishingly unlikely to prevent you from solving the problems that you need to solve.

Having said all that, when learning to program we *do* need to pick a language to work in, so we might as well pick one that's going to make the job easier. Python is such a language for a number of reasons:

- It has a mostly-consistent syntax, so you can generally learn one way of doing things and then apply it in multiple places
- It has a sensible set of built-in libraries for doing lots of common tasks
- It is designed in such a way that there's an obvious way of doing most things
- It's one of the most widely-used languages in the world, and there's a lot of advice, documentation and tutorials available on the web
- It's designed in a way that lets you start to write useful programs as soon as possible

- Its use of indentation, while annoying to people who aren't used to it, is great for beginners as it enforces a certain amount of readability

Python also has a couple of points to recommend it to biologists and scientists specifically:

- It's widely used in the scientific community
- It has a couple of very well-designed libraries for doing complex scientific computing (although we won't encounter them in this book)
- It lend itself well to being integrated with other, existing tools
- It has features which make it easy to manipulate strings of characters (for example, strings of DNA bases and protein amino acid residues, which we as biologists are particularly fond of)

Python vs. Perl

For biologists, the question *"what language should I learn"* often really comes down to the question *"should I learn Perl or Python?"*, so let's answer it head on. Perl and Python are both perfectly good languages for solving a wide variety of biological problems. However, after extensive experience teaching both Perl and Python to biologists, I've come the conclusion that Python is an easier language to learn by virtue of being more **consistent** and more **readable**.

An important thing to understand about Perl and Python is that they are *incredibly* similar (despite the fact that they look very different), so the point above about learning a second language applies doubly. Many Python and Perl features have a one-to-one correspondence, and so learning Perl after learning Python will be relatively easy – much easier than, for example, moving to Java or C.

How to use this book

Programming books generally fall into two categories; reference-type books, which are designed for looking up specific bits of information, and tutorial-type books, which are designed to be read cover-to-cover. This book is an example of the latter – code samples in later chapters often use material from previous ones, so you need to make sure you read the chapters in order. Exercises or examples from one chapter are sometimes used to illustrate the need for features that are introduced in the next.

There are a number of fundamental programming concepts that are relevant to material in multiple different chapters. In this book, rather than introduce these concepts all in one go, I've tried to explain them as they become necessary. This results in a tendency for earlier chapters to be longer than later ones, as they involve the introduction of more new concepts.

A certain amount of jargon is necessary if we want to talk about programs and programming concepts. I've tried to define each new technical term at the point where it's introduced, and then use it thereafter with occasional reminders of the meaning.

Chapters tend to follow a predictable structure. They generally start with a few paragraphs outlining the motivation behind the features that it will cover – why do they exist, what problems do they allow us to solve, and why are they useful in biology specifically? These are followed by the main body of the chapter in which we discuss the relevant features and how to use them. The length of the chapters varies quite a lot – sometimes we want to cover a topic briefly, other times we need more depth. This section ends with a brief recap outlining what we have learned, followed by exercises and solutions (more on that topic below).

A couple of notes on typography: **bold type** is used to emphasize important points and *italics* for technical terms and file names. Where code is mixed in with normal text it's written in a `mono-spaced font like this`. Occasionally there are footnotes[1] to provide additional information that is interesting to know but not crucial to understanding, or to give links to web pages.

Example code is highlighted with a solid border:

```
Some example code goes here
```

and example output (i.e. what we see on the screen when we run the code) is highlighted with a dotted border:

```
Some output goes here
```

Often we want to look at the code and the output it produces together. In these situations, you'll see a red-bordered code block followed immediately by a blue-bordered output block.

Sometimes it's necessary to refer in the text to individual lines of code or output, in which case I've used line numberings on the left:

```
1 first line
2 second line
3 third line
```

Other blocks of text (usually file contents or typed command lines) don't have any kind of border and look like this:

```
contents of a file
```

[1] Like this.

Exercises and solutions

The final part of each chapter is a set of exercises and solutions. The number and complexity of exercises differ greatly between chapters depending on the nature of the material. As a rule, early chapters have a large number of simple exercises, while later chapters have a small number of more complex ones. Many of the exercise problems are written in a deliberately vague manner and the exact details of how the solutions work is up to you (very much like real-life programming!) You can always look at the solutions to see one possible way of tackling the problem, but there are often multiple valid approaches.

I strongly recommend that you try tackling the exercises yourself before reading the solutions; there really is no substitute for practical experience when learning to program. I also encourage you to adopt an attitude of curious experimentation when working on the exercises – if you find yourself wondering if a particular variation on a problem is solvable, or if you recognize a closely-related problem from your own work, try solving it! Continuous experimentation is a key part of developing as a programmer, and the quickest way to find out what a particular function or feature will do is to try it.

The example solutions to exercises are written in a different way to most programming textbooks: rather than simply present the finished solution, I have outlined the thought processes involved in solving the exercises and shown how the solution is built up step-by-step. Hopefully this approach will give you an insight into the problem-solving mindset that programming requires. It's probably a good idea to read through the solutions even if you successfully solve the exercise problems yourself, as they sometimes suggest an approach that is not immediately obvious.

Getting in touch

One of the most convincing arguments for presenting a course like this one in the form of an ebook is that it can be continually updated and tweaked based on reader feedback. So, if you find anything that is hard to understand, or you think may contain an error, please get in touch – just drop me an email at martin@pythonforbiologists.com and I promise to get back to you.

Setting up your environment

All that you need in order to follow the examples and exercises in this book is a standard Python installation and a text editor. All the code in this book will run on either Linux, Mac or Windows machines. The slight differences between operating systems are explained in the text (mostly in chapter 9). If you have a choice of operating systems on which to learn Python, I recommend Linux, Mac OSX and Windows in that order, simply because the UNIX-based operating systems (Linux and OSX) are more amenable to programming in general.

Installing Python

The process of installing Python depends on the type of computer you're running on. If you're running a mainstream Linux distribution like Ubuntu, Python is probably already installed. To find out, open a terminal and type

```
 python
```

If you see some output along these lines:

```
Python 2.7.3 (default, Apr 10 2013, 05:13:16)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Then you are ready to go. If your Linux installation doesn't already have Python installed, try installing it with your package manager (the command will probably be either `sudo apt-get install python` or `sudo yum install python`). If this doesn't work, then download the package from the Python download page[1].

The official Python website has installation instructions for Mac[2] and Windows[3] computers as well; these are likely to be the most up-to-date instructions, so follow them closely.

Running Python programs

A Python program is just a normal text file that contains Python code. To run it we must first open up a command line. On Linux and Mac computers, the application to do this will be called something along the lines of "terminal". On Windows, it is known as "command prompt".

To run a Python program, we just type the path to the Python executable followed by the name of the file that contains the code we want to run[4]. On a Linux or Mac machine, the path will be something like:

```
/usr/local/bin/python
```

On Windows, it will be something like:

```
c:\Python27\python
```

[1] http://www.python.org/getit/

[2] http://www.python.org/getit/mac/

[3] http://www.python.org/getit/windows/

[4] When we refer to "a Python program" in this book, we are usually talking about the text file that holds the code.

To run a Python program, it's generally easiest to be in the same folder as it. By convention, Python programs are given the extension `.py`, so to run a program called test.py, we just type:

```
/usr/local/bin/python test.py
```

There are a couple of tricks that can be useful when experimenting with programs[1]. Firstly, you can run Python in an interactive (or "shell") mode by running it without the name of a program file. This allows you to type individual statements and see the result straight away.

Secondly, you can run Python with the `-i` option, which will cause it to run your program and **then** enter interactive mode. This can be handy if you want to examine the state of variables after your code has run.

Python 2 vs. Python 3

As will quickly become clear if you spend any amount of time on the official Python website, there are two versions of Python currently available. The Python world is, at the time of writing, in the middle of a transition from version 2 to version 3. A discussion of the pros and cons of each version is well beyond the scope of this book[2], but here's what you need to know: install Python 3 if possible, but if you end up with Python 2, don't worry – all the code examples in the book will work with both versions.

[1] Don't worry if these two options make no sense to you right now – they will do so later on in the book, once you've learned what statements and variables actually are.

[2] You might encounter writing online that makes the 2 to 3 changeover seem like a big deal, and it is – but only for existing, large projects. When writing code from scratch, as you'll be doing when learning, you're unlikely to run into any problems.

If you're going to use Python 2, there is just one thing that you have to do in order to make some of the code examples work: include this line at the start of all your programs:

```
from __future__ import division
```

We won't go into the explanation behind this line, except to say that it's necessary in order to correct a small quirk with the way that Python 2 handles division of numbers.

Depending on what version you use, you might see slight differences between the output in this book and the output you get when you run the code on your computer. I've tried to note these differences in the text where possible.

Text editors

Since a Python program is just a text file, you can create and edit it with any text editor of your choice. Note that by a text editor I **don't** mean a word processor – do **not** try to edit Python programs with Microsoft Word, LibreOffice Writer, or similar tools, as they tend to insert special formatting marks that Python cannot read.

When choosing a text editor, there is one feature that is essential[1] to have, and one which is nice to have. The essential feature is something that's usually called *tab emulation*. The effect of this feature at first seems quite odd; when enabled, it replaces any tab characters that you type with an equivalent number of space characters (usually set to four). The reason why this is useful is discussed at length in chapter 4, but here's a brief explanation: Python is very fussy about your use of tabs and spaces, and unless you are very disciplined when typing, it's easy to end up with a mixture of tabs and spaces in your programs. This causes very infuriating problems, because they look the same to you, but not to Python! Tab emulation

[1] OK, so it's not strictly essential, but you will find life much easer if you have it.

fixes the problem by making it effectively impossible for you to type a tab character.

The feature that is nice to have is *syntax highlighting*. This will apply different colours to different parts of your Python code, and can help you spot errors more easily.

Recommended text editors are **Notepad++** for Windows[1], **TextWrangler** for Mac OSX[2], and **gedit** for Linux[3], all of which are freely available.

On the web and elsewhere you may see references to Python IDEs. IDE stands for Integrated Development Environment, and they typically combine a text editor with a collection of other useful programming tools. While they can speed up development for experienced programmers, they're not a good idea for beginners as they complicate things, so I don't recommend you use them.

Reading the documentation

Part of the teaching philosophy that I've used in writing this book is that it's better to introduce a few useful features and functions rather than overwhelm you with a comprehensive list. The best place to go when you do want a complete list of the options available in Python is the official documentation[4] which, compared to many languages, is very readable.

[1] http://notepad-plus-plus.org/

[2] http://www.barebones.com/products/TextWrangler/

[3] https://projects.gnome.org/gedit/

[4] http://www.python.org/doc/

2: Printing and manipulating text

Why are we so interested in working with text?

Open the first page of a book about learning Python[1], and the chances are that the first examples of code you'll see involve **numbers**. There's a good reason for that: numbers are generally simpler to work with than text – there are not too many things you can do with them (once you've got basic arithmetic out of the way) and so they lend themselves well to examples that are easy to understand. It's also a pretty safe bet that the average person reading a programming book is doing so because they need to do some number-crunching.

So what makes this book different – why is this first chapter about text rather than numbers? The answer is that, as biologists, we have a particular interest in dealing with text rather than numbers (though of course, we'll need to learn how to manipulate numbers too). Specifically, we're interested in particular types of text that we call *sequences* – the DNA and protein sequences that constitute the data that we deal with in biology.

There are other reasons that we have a greater interest in working with text than the average novice programmer. As scientists, the programs that we write often need to work as part of a pipeline, alongside other programs that have been written by other people. To do this, we'll often need to write code that can **understand** the output from some other program (we call this *parsing*) or **produce** output in a format that another program can operate on. Both of these tasks require manipulating text.

I've hinted above that computers consider numbers and text to be different in some way. That's an important idea, and one that we'll return to in more detail later. For now, I want to introduce an important piece of jargon – the word *string*. String is

[1] Or indeed, any other programming language

the word we use to refer to a bit of text in a computer program (it just means a string of characters). From this point on we'll use the word *string* when we're talking about computer code, and we'll reserve the word *sequence* for when we're discussing biological sequences like DNA and protein.

Printing a message to the screen

The first thing we're going to learn is how to print[1] a message to the screen. Here's a line of Python code that will cause a friendly message to be printed. Quick reminder: solid lines indicate Python code, dotted lines indicate output.

```
print("Hello world")
```

Let's take a look at the various bits of this line of code, and give some of them names:

The whole line is called a *statement*.

`print` is the name of a *function*. The function tells Python, in vague terms, what we want to do – in this case, we want to print some text. The function name is always[2] followed by parentheses[3].

The bits of text inside the parentheses are called the *arguments* to the function. In this case, we just have one argument (later on we'll see examples of functions that take more than one argument, in which case the arguments are separated by commas).

[1] When we talk about printing text inside a computer program, we are not talking about producing a document on a printer. The word "print" is used for any occasion when our program outputs some text – in this case, the output is displayed in your terminal.

[2] This is not strictly true, but it's easier to just follow this rule than worry about the exceptions.

[3] There are several different types of brackets in Python, so for clarity we will always refer to *parentheses* when we mean these: (), *square brackets* when we mean these: [] and *curly brackets* when we mean these: {}

The arguments tell Python what we want to do more specifically – in this case, the argument tells Python exactly what it is we want to print: a friendly greeting.

Assuming you've followed the instructions in chapter 1 and set up your Python environment, type the line of code above into your favourite text editor, save it, and run it. You should see a single line of output like this:

```
Hello world
```

Quotes are important

In normal writing, we only surround a bit of text in quotes when we want to show that they are being said by somebody. In Python, however, strings are **always** surrounded by quotes. That is how Python is able to tell the difference between the instructions (like the function name) and the data (the thing we want to print). We can use either single or double quotes for strings – Python will happily accept either. The following two statements behave exactly the same:

```
print("Hello world")
print('Hello world')
```

Let's take a look at the output to prove it[1]:

```
Hello world
Hello world
```

You'll notice that the output above doesn't contain quotes – they are part of the code, not part of the string itself. If we **do** want to include quotes in the output, the easiest thing to do[2] is use the other type of quotes for surrounding the string:

[1] From this point on, I won't tell you to create a new file, enter the text, and run the program for each example – I will simply show you the output – but I encourage you to try the examples yourself.

[2] The alternative is to place a backslash character (\) before the quote – this is called *escaping* the quote and

```
print("She said, 'Hello world'")
print('He said, "Hello world"')
```

The above code will give the following output:

```
She said, 'Hello world'
He said, "Hello world"
```

Be careful when writing and reading code that involves quotes – you have to make sure that the quotes at the beginning and end of the string match up.

Use comments to annotate your code

Occasionally, we want to write some text in a program that is for humans to read, rather than for the computer to execute. We call this type of line a *comment*. To include a comment in your source code, start the line with a hash symbol[1]:

```
# this is a comment, it will be ignored by the computer
print("Comments are very useful!")
```

You're going to see a lot of comments in the source code examples in this book, and also in the solutions to the exercises. Comments are a very useful way to document your code, for a number of reasons:

- You can put the explanation of what a particular bit of code does right next to the code itself. This makes it much easier to find the documentation for a line of code that is in the middle of a large program, without having to search through a separate document.

will prevent Python from trying to interpret it.

[1] This symbol has many names – you might know it as number sign, pound sign, octothorpe, sharp (from musical notation), cross, or pig-pen.

- Because the comments are part of the source code, they can never get mixed up or separated. In other words, if you are looking at the source code for a particular program, then you automatically have the documentation as well. In contrast, if you keep the documentation in a separate file, it can easily become separated from the code.
- Having the comments right next to the code acts as a reminder to update the documentation whenever you change the code. The only thing worse than undocumented code is code with old documentation that is no longer accurate!

Don't make the mistake, by the way, of thinking that comments are only useful if you are planning on showing your code to somebody else. When you start writing your own code, you will be amazed at how quickly you forget the purpose of a particular section or statement. If you are working on a solution to one of the exercises in this book on Friday afternoon, then come back to it on Monday morning, it will probably take you quite a while to pick up where you left off.

Comments can help with this problem by giving you hints about the purpose of code, meaning that you spend less time trying to understand your old code, thus speeding up your progress. A side benefit is that writing a comment for a bit of code reinforces your understanding at the time you are doing it. A good habit to get into is writing a quick one-line comment above any line of code that does something interesting:

```
# print a friendly greeting
print("Hello world")
```

You'll see this technique used a lot in the code examples in this book, and I encourage you to use it for your own code as well.

Error messages and debugging

It may seem depressing early in the book to be talking about errors! However, it's worth pointing out at this early stage that **computer programs almost never work correctly the first time**. Programming languages are not like natural languages – they have a very strict set of rules, and if you break any of them, the computer will not attempt to guess what you intended, but instead will stop running and present you with an error message. You're going to be seeing a lot of these error messages in your programming career, so let's get used to them as soon as possible.

Forgetting quotes

Here's one possible error we can make when printing a line of output – we can forget to include the quotes:

```
print(Hello world)
```

This is easily done, so let's take a look at the output we'll get if we try to run the above code[1]:

```
$ python error.py
  File "error.py", line 1
    print(Hello world)
                    ^
SyntaxError: invalid syntax
```

[1] The output that you see might be very slightly different from this, depending on a bunch of factors like your operating system and the exact version of Python you are using.

Referring to the line numbers on the left we can see that the name of the Python file is `error.py` (line 1) and that the error occurs on the first line of the file (line 2). Python's best guess at the location of the error is just before the close parentheses (line 3). Depending on the type of error, this can be wrong by quite a bit, so don't rely on it too much!

The type of error is a `SyntaxError` (line 5), which mean that Python can't understand the code – it breaks the rules in some way (in this case, the rule that strings must be surrounded by quotation marks). We'll see different types of errors later in this book.

Spelling mistakes

What happens if we miss-spell the name of the function?:

```
prin("Hello world")
```

We get a different type of error – a `NameError` – and the error message is a bit more helpful:

```
1 $ python error.py
2 Traceback (most recent call last):
3   File "error.py", line 1, in <module>
4     prin("Hello world")
5 NameError: name 'prin' is not defined
```

This time, Python doesn't try to show us where on the line the error occurred, it just shows us the whole line (line 4). The error message tells us which word Python doesn't understand (line 5), so in this case, it's quite easy to fix.

Splitting a statement over two lines

What if we want to print some output that spans multiple lines? For example, we want to print the word "Hello" on one line and then the word "World" on the next line – like this:

```
Hello
World
```

We might try putting a new line in the middle of our string like this:

```
print("Hello
World")
```

but that won't work and we'll get the following error message:

```
1 $ python error.py
2   File "error.py", line 1
3     print("Hello
4                 ^
5 SyntaxError: EOL while scanning string literal
```

Python finds the error when it gets to the end of the first line of code (line 2 in the output). The error message (line 5) is a bit more cryptic than the others. *EOL* stands for End Of Line, and *string literal* means a string in quotes. So to put this error message in plain English: *"I started reading a string in quotes, and I got to the end of the line before I came to the closing quotation mark"*

If splitting the line up doesn't work, then how do we get the output we want.....?

Printing special characters

The reason that the code above didn't work is that Python got confused about whether the new line was part of the *string* (which is what we wanted) or part of the *source code* (which is how it was actually interpreted). What we need is a way to include a new line as part of a string, and luckily for us, Python has just such a tool built in. To include a new line, we write a backslash followed by the letter n – Python knows that this is a special character and will interpret it accordingly. Here's the code which prints "Hello world" across two lines:

```
# how to include a new line in the middle of a string
print("Hello\nworld")
```

Notice that there's no need for a space before or after the new line.

There are a few other useful special characters as well, all of which consist of a backslash followed by a letter. The only ones which you are likely to need for the exercises in this book are the *tab* character (`\t`) and the *carriage return* character (`\r`). The tab character can sometimes be useful when writing a program that will produce a lot of output. The carriage return character works a bit like a new line in that it puts the cursor back to the start of the line, but doesn't actually start a new line, so you can use it to overwrite output – this is sometimes useful for long-running programs.

Storing strings in variables

OK, we've been playing around with the `print` function for a while; let's introduce something new. We can take a string and assign a name to it using an equals sign – we call this a *variable*:

```
# store a short DNA sequence in the variable my_dna
my_dna = "ATGCGTA"
```

The variable `my_dna` now points to the string `"ATGCGTA"`. We call this *assigning* a variable, and once we've done it, we can use the variable name instead of the string itself – for example, we can use it in a `print` statement[1]:

```
# store a short DNA sequence in the variable my_dna
my_dna = "ATGCGTA"
# now print the DNA sequence
print(my_dna)
```

Notice that when we use the variable in a `print` statement, we don't need any quotation marks – the quotes are part of the string, so they are already "built in" to the variable `my_dna`.

We can change the value of a variable as many times as we like once we've created it:

```
my_dna = "ATGCGTA"
print(my_dna)
# change the value of my_dna
my_dna = "TGGTCCA"
```

Here's a very important point that trips many beginners up: variable names are **arbitrary** – that means that we can pick **whatever we like** to be the name of a variable. So our code above would work in exactly the same way if we picked a different variable name:

```
# store a short DNA sequence in the variable banana
banana = "ATGCGTA"
# now print the DNA sequence
print(banana)
```

[1] If it's not clear why this is useful, don't worry – it will become much more apparent when we look at some longer examples.

What makes a good variable name? Generally, it's a good idea to use a variable name that gives us a clue as to what the variable refers to. In this example, `my_dna` is a good variable name, because it tells us that the content of the variable is a DNA sequence. Conversely, `banana` is a bad variable name, because it doesn't really tell us anything about the value that's stored. As you read through the code examples in this book, you'll get a better idea of what constitutes good and bad variable names.

This idea – that names for things are arbitrary, and can be anything we like – is a theme that will occur many times in this book, so it's important to keep it in mind. Occasionally you will see a variable name that **looks like** it has some sort of relationship with the value it points to:

```
my_file = "my_file.txt"
```

but don't be fooled! Variable names and strings are separate things.

I said above that variable names can be anything we want, but it's actually not quite that simple – there are some rules we have to follow. We are only allowed to use letters, numbers, and underscores, so we can't have variable names that contain odd characters like £, ^ or %. We are not allowed to start a name with a number (though we can use numbers in the middle or at the end of a name). Finally, we can't use a word that's already built in to the Python language like "print".

It's also important to remember that variable names are case-sensitive, so `my_dna,  MY_DNA, My_DNA and My_Dna` are all separate variables. Technically this means that you could use all four of those names in a Python program to store different values, but please don't do this – it is very easy to become confused when you use very similar variable names.

Tools for manipulating strings

Now we know how to store and print strings, we can take a look at a few of the facilities that Python has for manipulating them. Python has many built-in tools for carrying out common operations, and in this next section we'll take a look at them one-by-one. In the exercises at the end of this chapter, we'll look at how we can use multiple different tools together in order to carry out more complex operations.

Concatenation

We can concatenate (stick together) two strings using the + symbol[1]. This symbol will join together the string on the left with the string on the right:

```
my_dna = "AATT" + "GGCC"
print(my_dna)
```

Let's take a look at the output:

```
AATTGGCC
```

In the above example, the things being concatenated were strings, but we can also use variables that point to strings:

```
upstream = "AAA"
my_dna = upstream + "ATGC"
# my_dna is now "AAAATGC"
```

[1] We call this the *concatenation operator*.

We can even join multiple strings together in one go:

```
upstream = "AAA"
downstream = "GGG"
my_dna = upstream + "ATGC" + downstream
# my_dna is now "AAAATGCGGG"
```

It's important to realize that the result of concatenating two strings together is itself a string. So it's perfectly OK to use a concatenation inside a print statement:

```
print("Hello" + " " + "world")
```

As we'll see in the rest of the book, using one tool inside another is quite a common thing to do in Python.

Finding the length of a string

Another useful built-in tool in Python is the `len` function (`len` is short for length). Just like the `print` function, the `len` function takes a single argument (take a quick look back at when we were discussing the `print` function for a reminder about what arguments are) which is a string. However, the behaviour of the `len` function is quite different. Instead of outputting text to the screen, `len` outputs a value that can be stored – we call this the *return value*. In other words, if we write a program that uses `len` to calculate the length of a string, the program will run but we won't see any output:

```
# this line doesn't produce any output
len("ATGC")
```

If we want to actually use the return value, we need to store it in a variable, and then do something useful with it (like printing it):

```
dna_length = len("AGTC")
print(dna_length)
```

There's another interesting thing about the `len` function: the result (or *return value*) is not a string, it's a number. This is a very important idea so I'm going to write it out in bold: **Python treats strings and numbers differently.**

We can see that this is the case if we try to concatenate together a number and a string. Consider this short program which calculates the length of a DNA sequence and then prints a message telling us the length:

```
# store the DNA sequence in a variable
my_dna = "ATGCGAGT"
# calculate the length of the sequence and store it in a variable
dna_length = len(my_dna)
# print a message telling us the DNA sequence lenth
print("The length of the DNA sequence is " + dna_length)
```

When we try to run this program, we get the following error:

```
1 $ python error.py
2 Traceback (most recent call last):
3   File "error.py", line 6, in <module>
4     print("The length of the DNA sequence is " + dna_length)
5 TypeError: cannot concatenate 'str' and 'int' objects
```

The error message (line 5) is short but informative: "`cannot concatenate 'str' and 'int' objects`". Python is complaining that it doesn't know how to concatenate a string (which it calls `str` for short) and a number (which it calls `int` – short for integer). Strings and numbers are examples of *types* – different kinds of information that can exist inside a program.

Happily, Python has a built-in solution – a function called `str` which turns a number[1] into a string so that we can print it. Here's how we can modify our program to use it – I've removed the comments from this version to make it a bit more compact:

```
my_dna = "ATGCGAGT"
dna_length = len(my_dna)
print("The length of the DNA sequence is " + str(dna_length))
```

The only thing we have changed is that we've replace `dna_length` with `str(dna_length)` inside the `print` statement[2]. Notice that because we're using one function (`str`) inside another function (`print`), our statement now ends with two closing parentheses.

To finish our discussion of the `str` function, here's a formal description of it, with all the technical terms in italics:

`str` is a *function* which takes one *argument* (whose type is *number*), and *returns* a value (whose type is *string*) representing that number.

If you're unsure about the meanings of any of the words in italics, skip back to the earlier parts of this chapter where we discussed them. Understanding how types work is key to avoiding many of the frustrations which new programmers typically encounter, so make sure the idea is clear in your mind before moving on with the rest of this book.

Changing case

We can convert a string to lower case by using a new type of syntax – a *method* that belongs to strings. A *method* is like a *function*, but instead of being built in to the

[1] Or a value of any non-string type, but we'll come to that later

[2] If you experiment with some of the code here, you might discover that you can also print a number directly without using `str` – but only if you don't try to concatenate it.

Python language, it belongs to a particular *type*. The method we are talking about here is called `lower`, and we say that it belongs to the *string* type. Here's how we use it:

```
my_dna = "ATGC"
# print my_dna in lower case
print(my_dna.lower())
```

Notice how using a method looks different to using a function. When we use a function like `print` or `len`, we write the function name first and the arguments go in parentheses:

```
print("ATGC")
len(my_dna)
```

When we use a method, we write the name of the variable first, followed by a period, then the name of the method, then the method arguments in parentheses. For the example we're looking at here, `lower`, there is no argument, so the opening and closing parentheses are right next to each other.

It's important to notice that the `lower` method does not actually change the variable; instead it returns a copy of the variable in lower case. We can prove that it works this way by printing the variable before and after running `lower`. Here's the code to do so:

```
my_dna = "ATGC"
# print the variable
print("before: " + my_dna)
# run the lower method and store the result
lowercase_dna = my_dna.lower()
# print the variable again
print("after: " + my_dna)
```

and here's the output we get:

```
before: ATGC
after: ATGC
```

Just like the `len` function, in order to actually do anything useful with the `lower` method, we need to store the result (or print it right away).

Because the `lower` method belongs to the string type, we can only use it on variables that are strings. If we try to use it on a number:

```
my_number = len("AGTC")
# my_number is 4
print(my_number.lower())
```

we will get an error that looks like this:

```
AttributeError: 'int' object has no attribute 'lower'
```

The error message is a bit cryptic, but hopefully you can grasp the meaning: something that is a number (an `int`, or integer) does not have a `lower` method. This is a good example of the importance of types in Python code: **we can only use methods on the type that they belong to**.

Before we move on, let's just mention that there is another method that belongs to the string type called `upper` – you can probably guess what it does!

Replacement

Here's another example of a useful method that belongs to the string type: `replace`. `replace` is slightly different from anything we've seen before – it takes two arguments (both strings) and returns a copy of the variable where all occurrences of the first string are replaced by the second string. That's quite a long-winded description, so here are a few examples to make things clearer:

```
protein = "vlspadktnv"
# replace valine with tyrosine
print(protein.replace("v", "y"))
# we can replace more than one character
print(protein.replace("vls", "ymt"))
# the original variable is not affected
print(protein)
```

And this is the output we get:

```
ylspadktny
ymtpadktnv
vlspadktnv
```

We'll take a look at more tools for carrying out string replacement in chapter 7.

Extracting part of a string

What do we do if we have a long string, but we only want a short portion of it? This is known as taking a *substring*, and it has its own notation in Python. To get a substring, we follow the variable name with a pair of square brackets which enclose a start and stop position, separated by a colon. Again, this is probably easier to visualize with a couple of examples – let's reuse our protein sequence from before:

```
protein = "vlspadktnv"
# print positions three to five
print(protein[3:5])
# positions start at zero, not one
print(protein[0:6])
# if we use a stop position beyond the end, it's the same as using the end
print(protein[0:60])
```

and here's the output:

```
pa
vlspad
vlspadktnv
```

There are two important things to notice here. Firstly, we actually start counting from position zero, rather than one – in other words, position 3 is actually the fourth character[1]. This explains why the first character of the first line of output is `p` and not `s` as you might think. Secondly, the positions are **inclusive** at the start, but **exclusive** at the stop. In other words, the expression `protein[3:5]` gives us everything starting at the third character, and stopping just before the fifth character (i.e. characters three and four).

If we just give a single number in the square brackets, we'll just get a single character:

```
protein = "vlspadktnv"
first_residue = protein[0]
```

We'll learn a lot more about this type of notation, and what we can do with it, in chapter 4.

Counting and finding substrings

A very common job in biology is to count the number of times some pattern occurs in a DNA or protein sequence. In computer programming terms, what that translates to is counting the number of times a *substring* occurs in a *string*. The method that does the job is called `count`. It takes a single argument whose type is string, and returns the number of times that the argument is found in the variable. The return type is a number, so be careful about how you use it!

[1] This seems very annoying when you first encounter it, but we'll see later why it's necessary.

Let's use our protein sequence one last time as an example. Remember that we have to use our old friend `str` to turn the counts into strings so that we can print them. Also, notice that here I have used a blank line to separate out the two bits of the program (calculating the counts, and printing them). Python is perfectly happy with this – it just ignores blank lines, so it's fine to put them in in order to make your programs more readable for humans.

```
protein = "vlspadktnv"
# count amino acid residues
valine_count = protein.count('v')
lsp_count = protein.count('lsp')
tryptophan_count = protein.count('w')

# now print the counts
print("valines: " + str(valine_count))
print("lsp: " + str(lsp_count))
print("tryptophans: " + str(tryptophan_count))
```

The output shows how the count method behaves:

```
valines: 2
leucines: 1
tryptophans: 0
```

A closely-related problem to counting substrings is finding their location. What if instead of counting the number of proline residues in our protein sequence we want to know where they are? The `find` method will give us the answer, at least for simple cases. `find` takes a single string argument, just like `count`, and returns a number which is the position at which that substring first appears in the string (in computing, we call that the *index* of the substring).

Remember that in Python we start counting from zero rather than one, so position 0 is the first character, position 4 is the fifth character, etc. A couple of examples:

```
protein = "vlspadktnv"
print(str(protein.find('p')))
print(str(protein.find('kt')))
print(str(protein.find('w')))
```

And the output:

```
3
6
-1
```

Notice the behaviour of find when we ask it to locate a substring that doesn't exist – we get back the answer `-1`.

Both `count` and `find` have a pretty serious limitation: you can only search for exact substrings. If you need to count the number of occurrences of a variable protein motif, or find the position of a variable transcription factor binding site, they will not help you. The whole of chapter 7 is devoted to tools that can do those kinds of jobs.

Of the tools we've discussed in this section, three – `replace`, `count` and `find` – require at least two strings to work, so be careful that you don't get confused about the order – remember that:

```
my_dna.count(my_motif)
```

is **not** the same as:

```
my_motif.count(my_dna)
```

Splitting up a string into multiple bits

An obvious question which biologists often ask when learning to program is "how do we split a string (e.g. a DNA sequence) into multiple pieces?" That's a common job in biology, but unfortunately we can't do it yet using the tools from this chapter. We'll talk about various different ways of splitting strings in chapter 4. I mention it here just to reassure you that we will learn how to do it eventually!

Recap

We started this chapter talking about strings and how to work with them, but along the way we had to take a lot of diversions, all of which were necessary to understand how the different string tools work. Thankfully, that means that we've covered most of the nuts and bolts of the Python language, which will make future chapters go much more smoothly.

We've learned about some general features of the Python programming language like

- the difference between *functions*, *statements* and *arguments*
- the importance of *comments* and how to use them
- how to use Python's error messages to fix bugs in our programs
- how to store *values* in *variables*
- the way that *types* work, and the importance of understanding them
- the difference between *functions* and *methods*, and how to use them both

And we've encountered some tools that are specifically for working with strings:

- concatenation
- different types of quotes and how to use them
- special characters

- changing the case of a string
- finding and counting substrings
- replacing bits of a string with something new
- extracting bits of a string to make a new string

Many of the above topics will crop up again in future chapters, and will be discussed in more detail, but you can always return to this chapter if you want to brush up on the basics. The exercises for this chapter will allow you to practice using the string manipulation tools and to become familiar with them. They'll also give you the chance to practice builder bigger programs by using the individual tools as building blocks.

Exercises

Reminder: the descriptions of the exercises are deliberately terse and may be somewhat ambiguous (just like requirements for programs you will write in real life). See the solutions for in-depth discussions of the exercises.

Calculating AT content

Here's a short DNA sequence:

```
ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT
```

Write a program that will print out the AT content of this DNA sequence. Hint: you can use normal mathematical symbols like add (+), subtract (-), multiply (*), divide (/) and parentheses to carry out calculations on numbers in Python.

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your program:

```
from __future__ import division
```

Complementing DNA

Here's a short DNA sequence:

```
ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT
```

Write a program that will print the complement of this sequence.

Restriction fragment lengths

Here's a short DNA sequence:

```
ACTGATCGATTACGTATAGTAGAATTCTATCATACATATATATCGATGCGTTCAT
```

The sequence contains a recognition site for the EcoRI restriction enzyme, which cuts at the motif G*AATTC (the position of the cut is indicated by an asterisk). Write a program which will calculate the size of the two fragments that will be produced when the DNA sequence is digested with EcoRI.

Splicing out introns, part one

Here's a short section of genomic DNA:

ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATCGATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT

It comprises two exons and an intron. The first exon runs from the start of the sequence to the sixty-third character, and the second exon runs from the ninety-first character to the end of the sequence. Write a program that will print just the coding regions of the DNA sequence.

Splicing out introns, part two

Using the data from part one, write a program that will calculate what percentage of the DNA sequence is coding.

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your program:

```
from __future__ import division
```

Splicing out introns, part three

Using the data from part one, write a program that will print out the original genomic DNA sequence with coding bases in uppercase and non-coding bases in lowercase.

Solutions

Calculating AT content

This exercise is going to involve a mixture of strings and numbers. Let's remind ourselves of the formula for calculating AT content:

$$AT\ content = \frac{A+T}{length}$$

There are three numbers we need to figure out: the number of `A`s, the number of `T`s, and the length of the sequence. We know that we can get the length of the sequence using the `len` function, and we can count the number of `A`s and `T`s using the `count` method. Here are a few lines of code that we think will calculate the numbers we need:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
length = len(my_dna)
a_count = my_dna.count('A')
t_count = my_dna.count('T')
```

At this point, it seems sensible to check these lines before we go any further. So rather than diving straight in and doing some calculations, let's print out these numbers so that we can eyeball them and see if they look approximately right. We'll have to remember to turn the numbers into strings using `str` so that we can print them:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
length = len(my_dna)
a_count = my_dna.count('A')
t_count = my_dna.count('T')

print("length: " + str(length))
print("A count: " + str(a_count))
print("T count: " + str(t_count))
```

Let's take a look at the output from this program:

```
length: 54
A count: 16
T count: 21
```

That looks about right, but how do we know if it's exactly right? We could go through the sequence manually base by base, and verify that there are sixteen As and eighteen Ts, but that doesn't seem like a great use of our time: also, what would we do if the sequence were 51 kilobases rather than 51 bases? A better idea is to run the exact same code with a much shorter test sequence, to verify that it works before going ahead and running it on the larger sequence.

Here's a version that uses a very short test sequence with one of each of the four bases:

```
test_dna = "ATGC"
length = len(test_dna)
a_count = test_dna.count('A')
t_count = test_dna.count('T')

print("length: " + str(length))
print("A count: " + str(a_count))
print("T count: " + str(t_count))
```

and here's the output:

```
length: 4
A count: 1
T count: 1
```

Everything looks OK – we can probably go ahead and run the code on the long sequence. But wait; we know that the next step is going to involve doing some calculations using the numbers. If we switch back to the long sequence now, then we'll be in the same position as we were before – we'll end up with an answer for the AT content, but we won't know if it's the right one.

A better plan is to stick with the short test sequence until we've written the whole program, and check that we get the right answer for the AT content (we can easily see by glancing at the test sequence that the AT content is 0.5). Here goes – we'll use the add and divide symbols from the exercise hint:

```
test_dna = "ATGC"
length = len(test_dna)
a_count = test_dna.count('A')
t_count = test_dna.count('T')

at_content = a_count + t_count / length
print("AT content is " + str(at_content))
```

The output from this program looks like this:

```
AT content is 1.25
```

That doesn't look right. Looking back at the code we can see what has gone wrong – in the calculation, the division has taken precedence over the addition, so what we have actually calculated is:

$$A+\frac{T}{length}$$

To fix it, all we need to do is add some parentheses around the addition, so that the line becomes:

```
at_content = (a_count + t_count) / length
```

Now we get the correct output for the test sequence:

```
AT content is 0.5
```

and we can go ahead and run the program using the longer sequence, confident that the code is working and that the calculations are correct. Here's the final version:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
length = len(my_dna)
a_count = my_dna.count('A')
t_count = my_dna.count('T')

at_content = (a_count + t_count) / length
print("AT content is " + str(at_content))
```

and the final output:

```
AT content is 0.6851851851851852
```

Complementing DNA

This one seems pretty straightforward – we need to take our sequence and replace A with T, T with A, C with G, and G with C. We'll have to make four separate calls to `replace`, and use the return value for each on as the input for the next tone. Let's try it:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
# replace A with T
replacement1 = my_dna.replace('A', 'T')
# replace T with A
replacement2 = replacement1.replace('T', 'A')
# replace C with G
replacement3 = replacement2.replace('C', 'G')
# replace G with C
replacement4 = replacement3.replace('G', 'C')
# print the result of the final replacement
print(replacement4)
```

When we take a look at the output, however, something seems wrong:

```
ACACAACCAAAACCAAAACAAAAACCAAACAAACAAAAAAAACCAACCCAACAA
```

We can see just by looking at the original sequence that the first letter is A, so the first letter of the printed sequence should be its complement, T. But instead the first letter is A. In fact, all of the bases in the printed sequence are either A or T. This is definitely not what we want!

Let's try and track the problem down by printing out all the intermediate steps as well:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
replacement1 = my_dna.replace('A', 'T')
print(replacement1)
replacement2 = replacement1.replace('T', 'A')
print(replacement2)
replacement3 = replacement2.replace('C', 'G')
print(replacement3)
replacement4 = replacement3.replace('G', 'C')
print(replacement4)
```

The output from this program makes it clear what the problem is:

```
TCTGTTCGTTTTCGTTTTGTTTTTGCTTTCTTTCTTTTTTTTCGTTGCGTTCTT
ACAGAACGAAAACGAAAAGAAAAAGCAAACAAACAAAAAAAACGAAGCGAACAA
AGAGAAGGAAAAGGAAAAGAAAAAGGAAAGAAAGAAAAAAAAGGAAGGGAAGAA
ACACAACCAAAACCAAAACAAAAACCAAACAAACAAAAAAAACCAACCCAACAA
```

The first replacement (the result of which is shown in the first line of the output) works fine – all the As have been replaced with Ts (for example, look at the first character – it's A in the original sequence and T in the first line of the output).

The second replacement is where it starts to go wrong: all the Ts are replaced by As, **including those that were there as a result of the first replacement**. So during the first two replacements, the first character is changed from A to T and then straight back to A again.

How are we going to get round this problem? One option is to pick a temporary alphabet of four letters and do each replacement twice:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
replacement1 = my_dna.replace('A', 'H')
replacement2 = replacement1.replace('T', 'J')
replacement3 = replacement2.replace('C', 'K')
replacement4 = replacement3.replace('G', 'L')
replacement5 = replacement4.replace('H', 'T')
replacement6 = replacement5.replace('J', 'A')
replacement7 = replacement6.replace('K', 'G')
replacement8 = replacement7.replace('L', 'C')
print(replacement8)
```

This gets us the result we are looking for. It avoids the problem with the previous program by using another letter to stand in for each base while the replacements are being done. For example, A is first converted to H and then later on H is converted to T.

Here's a slightly more elegant way of doing it. We can take advantage of the fact that the `replace` method is case-sensitive, and make all the replaced bases lower case. Then, once all the replacements have been carried out, we can simply call `upper` and change the whole sequence back to upper case. Let's take a look at how this works:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
replacement1 = my_dna.replace('A', 't')
print(replacement1)
replacement2 = replacement1.replace('T', 'a')
print(replacement2)
replacement3 = replacement2.replace('C', 'g')
print(replacement3)
replacement4 = replacement3.replace('G', 'c')
print(replacement4)
print(replacement4.upper())
```

The output lets us see exactly what's happening – notice that in this version of the program we print the final string twice, once as it is and then once converted to upper case:

```
tCTGtTCGtTTtCGTtTtGTtTTTGCTtTCtTtCtTtTtTtTCGtTGCGTTCtT
tCaGtaCGtaatCGatatGataaaGCataCtatCtatatataCGtaGCGaaCta
tgaGtagGtaatgGatatGataaaGgatagtatgtatatatagGtaGgGaagta
tgactagctaatgcatatcataaacgatagtatgtatatatagctacgcaagta
TGACTAGCTAATGCATATCATAAACGATAGTATGTATATATAGCTACGCAAGTA
```

We can see that as the program runs, each base in turn is replaced by its complement in lower case. Since the next replacement is only looking for upper case characters, bases don't get changed back as they did in the first version of our program.

Restriction fragment lengths

Let's start this exercise by solving the problem manually. If we look through the DNA sequence we can spot the EcoRI site at position 21. Here's the sequence with the base positions labelled above and the EcoRI motif in bold:

```
          1         2         3         4         5
0123456789012345678901234567890123456789012345678901234
ACTGATCGATTACGTATAGTAGAATTCTATCATACATATATATCGATGCGTTCAT
```

Since the EcoRI enzyme cuts the DNA between the G and first A, we can figure out that the first fragment will run from position 0 to position 21, and the second fragment from position 22 to the last position, 54. Therefore the lengths of the two fragments are 22 and 33.

Writing a program to figure out the lengths is just a question of applying the same logic. We'll use the `find` method to figure out the position of the start of the EcoRI motif, then add one to account for the fact that the positions start counting from zero – this will give us the length of the first fragment. From there we can get the length of the second fragment by finding the length of the input sequence and subtracting the length of the first fragment:

```
my_dna = "ACTGATCGATTACGTATAGTAGAATTCTATCATACATATATATCGATGCGTTCAT"
frag1_length = my_dna.find("GAATTC") + 1
frag2_length = len(my_dna) - frag1_length
print("length of fragment one is " + str(frag1_length))
print("length of fragment two is " + str(frag2_length))
```

The output from this program confirms that it agrees with the answer we got manually:

```
length of fragment one is 22
length of fragment two is 33
```

If we wanted to run the same program using a different restriction enzyme, we'd have to change **both** the string that we used in the `find` method call, **and** the number that we add in order to take account of the cut site.

It's worth noting that this program assumes that the DNA sequence definitely does contain the restriction site we're looking for. If we try the same program using a DNA sequence which doesn't contain the site, it will report a fragment of length 0 and a fragment whose length is equal to the total length of the DNA sequence. While this is not strictly wrong, it's a little misleading – if we were going to use this program for real-life work, we'd probably prefer to have slightly different behaviour depending on whether or not the DNA sequence contained the motif we're looking for. We'll talk about how to implement that type of behaviour in chapter 6.

Splicing out introns, part one

In this exercise, we're being asked to produce a program that does the job of a spliceosome – splits a DNA sequence at two specified locations to make three pieces, then join the outer two pieces together[1].

Let's start by splitting the sequence up into three bits. We'll have to use the substring notation from earlier in the chapter, and we'll need to take care with the numbers. We know that if we give a stop position for a substring then it will go on to the end of the input string, so rather than figure out the position of the end of the sequence, we'll just be lazy and use a big number. Here's the code (the first line, where we store the DNA sequence in the variable `my_dna`, is too long to fit on one line on the page, so it looks like it's spread out over multiple lines):

[1] We know that that's not really how a splicosome works, but it's fine as a conceptual model.

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATC
GATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[1:63]
exon2 = my_dna[91:10000]
print(exon1 + exon2)
```

The output from this code looks vaguely right:

```
TCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCATCGATCGAT
ATCGATGCATCGACTACTAT
```

but when we look more closely we can see that something is not right. The printed coding sequence is supposed to start at the very first character of the input sequence, but it's starting at the second. We have forgotten to take into account the fact that Python starts counting from zero, so our numbers are all too high by one. Let's try again:

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATC
GATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:62]
exon2 = my_dna[90:10000]
print(exon1 + exon2)
```

Now the output looks correct – the coding sequence starts at the very beginning of the input sequence:

```
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCATCGATCGA
TATCGATGCATCGACTACTAT
```

Splicing out introns, part two

This is a straightforward piece of number-crunching. There are a couple of ways to go about it. We could use the exon start-stop coordinates to calculate the length of the coding portion of the sequence. However, since we've already written the code to generate the coding sequence, we can simply calculate the length of it, and then divide by the length of the input sequence:

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATC
GATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:62]
exon2 = my_dna[90:10000]
coding_length = len(exon1 + exon2)
total_length = len(my_dna)
print(coding_length / total_length)
```

The output shows that we're nearly right:

```
0.7723577235772358
```

We have calculated the coding proportion as a fraction, but the exercise called for a percentage. We can easily fix this by multiplying by 100. Notice that the symbol for multiplication is not `x`, as you might think, but `*`. The final code:

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATC
GATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:62]
exon2 = my_dna[90:10000]
coding_length = len(exon1 + exon2)
total_length = len(my_dna)
print(100 * coding_length / total_length)
```

gives the correct output:

```
77.23577235772358
```

although we probably don't really require that number of significant figures. In chapter 5 we will learn how to format the output nicely.

Splicing out introns, part three

This sounds quite tricky, but we have already done the hard bit in part one. All we need to do is extract the intron sequence as well as the exons, convert it to lower case, then concatenate the three sequences to recreate the original genomic sequence:

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATC
GATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:62]
intron = my_dna[62:90]
exon2 = my_dna[90:10000]
print(exon1 + intron.lower() + exon2)
```

Looking at the output, we see an upper case DNA sequence with a lower case section in the middle, as expected:

```
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGatcgatcgatcg
atcgatcgatcatgctATCATCGATCGATATCGATGCATCGACTACTAT
```

When we are applying several transformations to text, as in this exercise, there are usually a number of different ways we can write the program. For example, we could store the lower case version of the intron, rather than converting it to lower case when printing:

```
intron = my_dna[62:90].lower()
```

Or we could avoid using variables for the introns and exons all together, and do everything in one big print statement:

```
print(my_dna[0:62] + my_dna[62:90].lower() + my_dna[90:10000])
```

This last option is very concise, but a bit harder to read than the more verbose way.

As the exercises in this book get longer, you'll notice that there are more and more different ways to write the code – you may end up with solutions that look very different to the example solutions. When trying to choose between different ways to write a program, always favour the solution that is clearest in intent and easiest to read.

3: Reading and writing files

Why are we so interested in working with files?

As we start this chapter, we find ourselves once again doing things in a slightly different order to most programming books. The majority of introductory programming books won't consider working with external files until much further along, so why are we introducing it now?

The answer, as was the case in the last chapter, lies in the particular jobs that we want to use Python for. The data that we as biologists work with is stored in files, so if we're going to write useful programs we need a way to get the data out of files and into our programs (and *vice versa*). As you were going through the exercises in the previous chapter, it may have occurred to you that copying and pasting a DNA sequence directly into a program each time we want to use it is not a very good approach to take, and you'd be right. The sequences we were working with in the exercises were very short; obviously real-life data will be much longer. Also, it seems inelegant to have the data we want to work on mixed up with the code that manipulates it. In this chapter we'll see a better way to do it.

We're lucky in biology that many of the types of data that we work with are stored in text[1] files which are relatively simple to process using Python. Chief among these, of course, are DNA and protein sequence data, which can be stored in a variety of formats.[2] But there are many other types of data – sequencing reads, quality scores, SNPs, phylogenetic trees, read maps, geographical sample data, genetic distance matrices – which we can access from within our Python programs.

[1] i.e. files which you can open in a text editor and read, as opposed to binary files which cannot be read directly.

[2] In this book we'll mostly be talking about FASTA format as it's the simplest and most common format, but there are many more.

Another reason for our interest in file input/output is the need for our Python programs to work as part of a pipeline or work flow involving other, existing tools. When it comes to using Python in the real world, we often want Python to either accept data from, or provide data to, another program. Often the easiest way to do this is to have Python read, or write, files in a format that the other program already understands.

Reading text from a file

Firstly, a quick note about what we mean by text. In programming, when we talk about *text files*, we are not necessarily talking about something that is human-readable. Rather, we are talking about a file that contains characters and lines – something that you could open up and view in a text editor, regardless of whether you could actually make sense of the file or not. Examples of text files which you might have encountered include:

- FASTA files of DNA or protein sequences
- files containing output from command-line programs (e.g. BLAST)
- FASTQ files containing DNA sequencing reads
- HTML files
- word processing documents
- and of course, Python code

In contrast, most files that you encounter day-to-day will be *binary files* – ones which are not made up of characters and lines, but of bytes. Examples include:

- image files (JPEGs and PNGs)
- audio files
- video files

- compressed files (e.g. ZIP files)

If you're not sure whether a particular file is text or binary, there's a very simple way to tell – just open it up in a text editor. If the file displays without any problem, then it's text (regardless of whether you can make sense of it or not). If you get an error or a warning from your text editor, or the file displays as a collection of indecipherable characters, then it's binary.

The examples and exercises in this chapter are a little different from those in the previous one, because they rely on the existence of the files that we are going to manipulate. If you want to try running the examples in this chapter, you'll need to make sure that there is a file in your working directory called *dna.txt* which has a single line containing a DNA sequence. The easiest way to do this is to run the examples while in the *chapter_3* folder inside the exercises download[1].

Using open to read a file

In Python, as in the physical world, we have to open a file before we can read what's inside it. The Python function that carries out the job of opening a file is very sensibly called `open`. It takes one argument – a string which contains the name of the file – and returns a *file object:*

```
my_file = open("dna.txt")
```

A *file object* is a new type which we haven't encountered before, and it's a little more complicated than the string and number types that we saw in the previous chapter. With strings and numbers it was easy to understand what they represented – a single bit of text, or a single number. A file object, in contrast, represents something a bit less tangible – it represents a file on your computer's hard drive.

[1] If you haven't downloaded the example files yet, you'll find the link in the email that came with your purchase of this book.

The way that we use file objects is a bit different to strings and numbers as well. If you glance back at the examples from the previous chapter you'll see that most of the time when we want to use a variable containing a string or number we just use the variable name:

```
my_string = 'abcdefg'
print(my_string)
my_number = 42
print(my_number + 1)
```

In contrast, when we're working with file objects most of our interaction will be through *methods*. This style of programming will seen unusual at first, but as we'll see in this chapter, the file type has a well thought-out set of methods which let us do lots of useful things.

The first thing we need to be able to do is to read the contents of the file. The file type has a `read` method which does this. It doesn't take any arguments, and the return value is a string, which we can store in a variable. Once we've read the file contents into a variable, we can treat them just like any other string – for example, we can print them:

```
my_file = open("dna.txt")
file_contents = my_file.read()
print(file_contents)
```

Files, contents and file names

When learning to work with files it's very easy to get confused between a *file object*, a *file name*, and the *contents* of a file. Take a look at the following bit of code:

```
1 my_file_name = "dna.txt"
2 my_file = open(my_file_name)
3 my_file_contents = my_file.read()
```

What's going on here? On line 1, we store the string *dna.txt* in the variable `my_file_name`. On line 2, we use the variable `my_file_name` as the argument to the `open` function, and store the resulting file object in the variable `my_file`. On line 3, we call the `read` method on the variable `my_file`, and store the resulting string in the variable `my_file_contents`.

The important thing to understand about this code is that there are three separate variables which have different types and which are storing three very different things. `my_file_name` is a string, and it stores the name of a file on disk. `my_file` is a file object, and it represents the file itself. `my_file_contents` is a string, and it stores the text that is in the file.

Remember that variable names are arbitrary – the computer doesn't care what you call your variables. So this piece of code is exactly the same as the previous example:

```
apple = "dna.txt"
banana = open(apple)
grape = banana.read()
```

except it is harder to read! In contrast, the file name (*dna.txt*) is **not** arbitrary – it must correspond to the name of a file on the hard drive of your computer.

A common error is to try to use the `read` method on the wrong thing. Recall that `read` is a method that only works on file objects. If we try to use the `read` method on the file name:

```
my_file_name = "dna.txt"
my_contents = my_file_name.read()
```

we'll get an `AttributeError` – Python will complain that strings don't have a `read` method[1]:

```
AttributeError: 'str' object has no attribute 'read'
```

Another common error is to use the *file object* when we meant to use the *file contents*. If we try to print the file object:

```
my_file_name = "dna.txt"
my_file = open(my_file_name)
print(my_file)
```

we won't get an error, but we'll get an odd-looking line of output:

```
<open file 'dna.txt', mode 'r' at 0x7fc5ff7784b0>
```

We won't discuss the meaning of this line now: just remember that if you try to print the contents of a file but instead you get some output that looks like the above, you have almost definitely printed the file object rather than the file contents.

Dealing with newlines

Let's take a look at the output we get when we try to print some information from a file. We'll use the *dna.txt* file from the *chapter_3* exercises folder. This file contains a single line with a short DNA sequence. Open the file up in a text editor and take a look at it.

We're going to write a simple program to read the DNA sequence from the file and print it out along with its length. Putting together the file functions and methods

[1] From now on, I'll just show the relevant bits of output when discussing error message.

from this chapter, and the material we saw in the previous chapter, we get the following code:

```
# open the file
my_file = open("dna.txt")
# read the contents
my_dna = my_file.read()
# calculate the length
dna_length = len(my_dna)
# print the output
print("sequence is " + my_dna +  " and length is " + str(dna_length))
```

When we look at the output, we can see that the program is working almost perfectly – but there is something strange: the output has been split over two lines:

```
sequence is ACTGTACGTGCACTGATC
 and length is 19
```

The explanation is simple once you know it: Python has included the new line character at the end of the *dna.txt* file as part of the contents. In other words, the variable `my_dna` has a new line character at the end of it. If we could view the `my_dna` variable directly[1], we would see that it looks like this:

```
 'ACTGTACGTGCACTGATC\n'
```

The solution is also simple. Because this is such a common problem, strings have a method for removing new lines from the end of them. The method is called `rstrip`, and it takes one string argument which is the character that you want to remove. In this case, we want to remove the newline character (`\n`). Here's a modified version of the code – note that the argument to `rstrip` is itself a string so needs to be enclosed in quotes:

[1] In fact, we can do this – there's a function called `repr` that returns a representation of a variable.

```
my_file = open("dna.txt")
my_file_contents = my_file.read()
# remove the newline from the end of the file contents
my_dna = my_file_contents.rstrip("\n")
dna_length = len(my_dna)
print("sequence is " + my_dna +  " and length is " + str(dna_length))
```

and now the output looks just as we expected:

```
sequence is ACTGTACGTGCACTGATC and length is 18
```

In the code above, we first read the file contents and then removed the newline, in two separate steps:

```
my_file_contents = my_file.read()
my_dna = my_file_contents.rstrip("\n")
```

but it's more common to read the contents and remove the newline all in one go, like this:

```
my_dna = my_file.read().rstrip("\n")
```

This is a bit tricky to read at first as we are using two different methods (`read` and `rstrip`) in the same statement. The key is to read it from left to right – we take the `my_file` variable and use the `read` method on it, then we take the output of that method (which we know is a string) and use the `rstrip` method on it. The result of the `rstrip` method is then stored in the `my_dna` variable.

If you find it difficult write the whole thing as one statement like this, just break it up and do the two things separately – your programs will run just as well.

Missing files

What happens if we try to read a file that doesn't exist?

```
my_file = open("nonexistent.txt")
```

We get a new type of error that we've not seen before:

```
IOError: [Errno 2] No such file or directory: 'nonexistent.txt'
```

Ideally, we'd like to be able to check if a file exists **before** we try to open it – we'll see how to do that in chapter 9.

Writing text to files

All the example programs that we've seen so far in this book have produced output straight to the screen. That's great for exploring new features and when working on programs, because it allows you to see the effect of changes to the code right away. It has a few drawbacks, however, when writing code that we might want to use in real life.

Printing output to the screen only really works well when there isn't very much of it[1]. It's great for short programs and status messages, but quickly becomes cumbersome for large amounts of output. Some terminals struggle with large amounts of text, or worse, have a limited scrollback capability which can cause the first bit of your output to disappear. It's not easy to search in output that's being displayed at the terminal, and long lines tend to get wrapped. Also, for many programs we want to send different bits of output to different files, rather than having it all dumped in the same place.

[1] Linux users may be aware that we can redirect terminal output to a file using shell redirection, which can get around some of these problems.

Most importantly, terminal output vanishes when you close your terminal program. For small programs like the examples in this book, that's not a problem – if you want to see the output again you can just re-run the program. If you have a program that requires a few hours to run, that's not such a great option.

Opening files for writing

In the previous section, we saw how to open a file and read its contents. We can also open a file and *write* some data to it, but we have to use the `open` function in a slightly different way. To open a file for writing, we use a two-argument version of the `open` function, where the second argument is a specially-formatted string describing what we want to do to the file[1]. This second argument can be "r" for reading, "w" for writing, or "a" for appending[2]. If we leave out the second argument (like we did for all the examples above), Python uses the default, which is "r" for reading.

The difference between "w" and "a" is subtle, but important. If we open a file that already exists using the mode "w", then we will overwrite the current contents with whatever data we write to it. If we open an existing file with the mode "a", it will add new data onto the end of the file, but will **not** remove any existing content. If there doesn't already exist a file with the specified name, then "w" and "a" behave identically – they will both create a new file to hold the output.

Quite a lot of Python functions and methods have these optional arguments. For the purposes of this book, we will only mention them when they are directly relevant to what we're doing. If you want to see all the optional arguments for a particular method or function, the best place to look is the official Python documentation – see chapter 1 for details.

[1] We call this the *mode* of the file.

[2] These are the most commonly-used options – there are a few others.

Once we've opened a file for writing, we can use the file `write` method to write some text to it. `write` works a lot like `print` – it takes a single string argument - but instead of printing the string to the screen it writes it to the file.

Here's how we use `open` with a second argument to open a file and write a single line of text to it:

```
my_file = open("out.txt", "w")
my_file.write("Hello world")
```

Because the output is being written to the file in this example, you won't see any output on the screen if you run it. To check that the code has worked, you'll have to run it, then open up the file *out.txt* in your text editor and check that its contents are what you expect[1].

Remember that with `write`, just like with `print`, we can use **any** string as the argument. This also means that we can use any method or function that **returns** a string. The following are all perfectly OK:

```
# write "abcdef"
my_file.write("abc" + "def")
# write "8"
my_file.write(str(len('AGTGCTAG')))
# write "TTGC"
my_file.write("ATGC".replace('A', 'T'))
# write "atgc"
my_file.write("ATGC".lower())
# write contents of my_variable
my_file.write(my_variable)
```

[1] .txt is the standard file name extension for a plain text file. Later in this book, when we generate output files with a particular format, we'll use different file name extensions.

Closing files

There's one more important file method to look at before we finish this chapter – `close`. Unsurprisingly, this is the opposite of `open` (but note that it's a *method*, whereas `open` is a *function*). We should call `close` after we're done reading or writing to a file – we won't go into the details here, but it's a good habit to get into as it avoids some types of bugs that can be tricky to track down[1]. `close` is an unusual method as it takes no arguments (so it's called with an empty pair of parentheses) and doesn't return any useful value:

```
my_file = open("out.txt", "w")
my_file.write("Hello world")
# remember to close the file
my_file.close()
```

Paths and folders

So far, we have only dealt with opening files in the same folder that we are running our program. What if we want to open a file from a different part of the file system?

The `open` function is quite happy to deal with files from anywhere on your computer, as long as you give the full path (i.e. the sequence of folder names that tells you the location of the file). Just give a *file path* as the argument rather than a *file name*. The format of the file path looks different depending on your operating system. If you're on Linux, it will look like this:

```
my_file = open("/home/martin/myfolder/myfile.txt")
```

if you're on Windows, like this[2]:

[1] Specifically, it helps to ensure that output to a file is flushed, which is necessary when we want to make a file available to another program as part of our work flow.

[2] The extra r character before the string is necessary to prevent Python from trying to interpret the backslash in the file path; see chapter 7 for an explanation.

```
my_file = open(r"c:\windows\Desktop\myfolder\myfile.txt")
```

and if you're on a Mac, like this:

```
my_file = open("/Users/martin/Desktop/myfolder/myfile.txt")
```

Recap

We've taken a whole chapter to introduce the various ways of reading and writing to files, because it's such an important part of building programs that are useful in biology. We've seen how working with file contents is always a two-step process – we must open a file before reading or writing – and looked at several common pitfalls. We'll return to the theme of file manipulation in later chapters where we'll address some of the shortcomings of the techniques we learned in this chapter:

- All the examples in this chapter than involve reading files do so by reading all the content into a single variable. Often, this is not what we want – a much more common requirement is to process a file line-by-line. In chapter 4 we'll learn about lists and loops, which will allow us to do exactly that.
- There are, of course, many things we want to do to files besides simply reading their contents. We would also like our programs to be able to move and copy files, to create and delete files and directories, and to list files and their properties. We'll cover the tools required to do these things in chapter 9.
- Finally, another feature common to all our examples is that the names of files are written as part of the code. We will generally want our real-life programs to be more flexible, and capable of reading files that are specified by the user. Chapter 9 also deals with the various forms of user input and in it we'll learn how to make our programs accept file names flexibly.

Exercises

Splitting genomic DNA

Look in the *chapter_3* folder for a file called *genomic_dna.txt* – it contains the same piece of genomic DNA that we were using in the final exercise from chapter 2. Write a program that will split the genomic DNA into coding and non-coding parts, and write these sequences to two separate files.

Hint: use your solution to the last exercise from chapter 2 as a starting point.

Writing a FASTA file

FASTA file format is a commonly-used DNA and protein sequence file format. A single sequence in FASTA format looks like this:

```
>sequence_name
ATCGACTGATCGATCGTACGAT
```

Where sequence_name is a header that describes the sequence (the greater-than symbol indicates the start of the header line). Often, the header contains an accession number that relates to the record for the sequence in a public sequence database. A single FASTA file can contain multiple sequences, like this:

```
>sequence_one
ATCGATCGATCGATCGAT
>sequence_two
ACTAGCTAGCTAGCATCG
>sequence_three
ACTGCATCGATCGTACCT
```

Write a program that will create a FASTA file for the following three sequences – make sure that all sequences are in upper case and only contain the bases A, T, G and C.

Sequence header	DNA sequence
ABC123	ATCGTACGATCGATCGATCGCTAGACGTATCG
DEF456	actgatcgacgatcgatcgatcacgact
HIJ789	ACTGAC-ACTGT--ACTGTA----CATGTG

Writing multiple FASTA files

Use the data from the previous exercise, but instead of creating a single FASTA file, create three new FASTA files – one per sequence. The names of the FASTA files should be the same as the sequence header names, with the extension *.fasta*.

Solutions

Splitting genomic DNA

We have a head-start on this problem, because we have already tackled a similar problem in the previous chapter. Let's remind ourselves of the solution we ended up with for that exercise:

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATC
GATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:62]
intron = my_dna[62:90]
exon2 = my_dna[90:10000]
print(exon1 + intron.lower() + exon2)
```

What changes do we need to make? Firstly, we need to read the DNA sequence from a file instead of writing it in the code:

```
dna_file = open("genomic_dna.txt")
my_dna = dna_file.read()
```

Secondly, we need to create two new file objects to hold the output:

```
coding_file = open("coding_dna.txt", "w")
noncoding_file = open("noncoding_dna.txt", "w")
```

Finally, we need to concatenate the two exon sequences and write them to the coding DNA file, and write the intron sequence to the non-coding DNA file:

```
coding_file.write(exon1 + exon2)
noncoding_file.write(intron)
```

Let's put it all together, with some blank lines to separate out the different parts of the program:

```
# open the file and read its contents
dna_file = open("genomic_dna.txt")
my_dna = dna_file.read()

# extract the different bits of DNA sequence
exon1 = my_dna[0:62]
intron = my_dna[62:90]
exon2 = my_dna[90:10000]

# open the two output files
coding_file = open("coding_dna.txt", "w")
noncoding_file = open("noncoding_dna.txt", "w")

# write the sequences to the output files
coding_file.write(exon1 + exon2)
noncoding_file.write(intron)
```

Writing a FASTA file

Let's start this problem by thinking about the variables we're going to need. We have three DNA sequences in total, so we'll need three variables to hold the sequence headers, and three more to hold the sequences themselves:

```
header_1 = "ABC123"
header_2 = "DEF456"
header_3 = "HIJ789"
seq_1 = "ATCGTACGATCGATCGATCGCTAGACGTATCG"
seq_2 = "actgatcgacgatcgatcgatcacgact"
seq_3 = "ACTGAC-ACTGT--ACTGTA----CATGTG"
```

FASTA format has alternating lines of header and sequence, so before we try any sequence manipulation, let's try to write a program that produces the lines in the right order. Rather than writing to a file, we'll print the output to the screen for now

– that will make it easier to see the output right away. Once we've got it working, we'll switch over to file output. Here's a few lines which will print data to the screen:

```
print(header_1)
print(seq_1)
print(header_2)
print(seq_2)
print(header_3)
print(seq_3)
```

and here's what the output looks like:

```
ABC123
ATCGTACGATCGATCGATCGCTAGACGTATCG
DEF456
actgatcgacgatcgatcgatcacgact
DEF456
actgatcgacgatcgatcgatcacgact
```

Not far off – the lines are in the right order, but we forgot to include the greater-than symbol at the start of the header. Also, we don't really need to print the header and the sequence separately for each sequence – we can include a newline character in the print string in order to get them on separate lines. Here's an improved version of the code:

```
print('>' + header_1 + '\n' + seq_1)
print('>' + header_2 + '\n' + seq_2)
print('>' + header_3 + '\n' + seq_3)
```

and the output looks better too:

```
>ABC123
ATCGTACGATCGATCGATCGCTAGACGTATCG
>DEF456
actgatcgacgatcgatcgatcacgact
>HIJ789
ACTGAC-ACTGT--ACTGTA----CATGTG
```

Next, let's tackle the problems with the sequences. The second sequence is in lower case, and it needs to be in upper case – we can fix that using the `upper` string method. The third sequence has a bunch of gaps that we need to remove. We haven't come across a remove method…. but we do know how to replace one character with another. If we replace all the gap characters with an empty string, it will be the same as removing them[1]. Here's a version that fixes both sequences:

```
print('>' + header_1 + '\n' + seq_1)
print('>' + header_2 + '\n' + seq_2.upper())
print('>' + header_3 + '\n' + seq_3.replace('-', ''))
```

Now the printed output is perfect:

```
>ABC123
ATCGTACGATCGATCGATCGCTAGACGTATCG
>DEF456
ACTGATCGACGATCGATCGATCACGACT
>HIJ789
ACTGACACTGTACTGTACATGTG
```

The final step is to switch from printed output to writing to a file. We'll `open` a new file, and change the three `print` lines to `write`:

[1] An empty string is just a pair of open and close quotation marks with nothing in between them.

```
output = open("sequences.fasta", "w")
output.write('>' + header_1 + '\n' + seq_1)
output.write('>' + header_2 + '\n' + seq_2.upper())
output.write('>' + header_3 + '\n' + seq_3.replace('-', ''))
```

After making these changes the code doesn't produce any output on the screen, so to see what's happened we'll need to take a look at the *sequences.fasta* file:

```
>ABC123
ATCGTACGATCGATCGATCGCTAGACGTATCG>DEF456
ACTGATCGACGATCGATCGATCACGACT>HIJ789
ACTGACACTGTACTGTACATGTG
```

This doesn't look right – the second and third lines have been joined together, as have the fourth and fifth. What has happened?

It looks like we've uncovered a difference between the `print` function and the `write` method. `print` automatically puts a new line at the end of the string, whereas `write` doesn't. This means we've got to be careful when switching between them! The fix is quite simple, we'll just add a newline onto the end of each string that gets written to the file:

```
output = open("sequences.fasta", "w")
output.write('>' + header_1 + '\n' + seq_1 + '\n')
output.write('>' + header_2 + '\n' + seq_2.upper() + '\n')
output.write('>' + header_3 + '\n' + seq_3.replace('-', '') + '\n')
```

The arguments for the write statements are getting quite complicated, but they are all made up of simple building blocks. For example the last one, if we translated it into English, would read "*a greater-than symbol, followed by the variable header_3, followed by a newline, followed by the variable seq_3 with all hyphens replaced with nothing, followed by another newline*".

Here's the final code, including the variable definition at the beginning, with blank lines and comments:

```
# set the values of all the header variables
header_1 = "ABC123"
header_2 = "DEF456"
header_3 = "HIJ789"

# set the values of all the sequence variables
seq_1 = "ATCGTACGATCGATCGATCGCTAGACGTATCG"
seq_2 = "actgatcgacgatcgatcgatcacgact"
seq_3 = "ACTGAC-ACTGT—ACTGTA----CATGTG"

# make a new file to hold the output
output = open("sequences.fasta", "w")

# write the header and sequence for seq1
output.write('>' + header_1 + '\n' + seq_1 + '\n')

# write the header and uppercase sequences for seq2
output.write('>' + header_2 + '\n' + seq_2.upper() + '\n')

# write the header and sequence for seq3 with hyphens removed
output.write('>' + header_3 + '\n' + seq_3.replace('-', '') + '\n')
```

Writing multiple FASTA files

We can solve this problem with a slight modification of our solution to the previous exercise. We'll need to create three new files to hold the output, and we'll construct the name of each file by using string concatenation:

```
output_1 = open(header_1 + ".fasta", "w")
output_2 = open(header_2 + ".fasta", "w")
output_3 = open(header_3 + ".fasta", "w")
```

Remember, the first argument to `open` is a string, so it's fine to use a concatenation because we know that the result of concatenating two strings is also a string.

We'll also change the `write` statements so that we have one for each of the output files. We need to be careful with the number here in order to make sure that we get the right sequence in each file. Here's the final code, with comments.

```
# set the values of all the header variables
header_1 = "ABC123"
header_2 = "DEF456"
header_3 = "HIJ789"

# set the values of all the sequence variables
seq_1 = "ATCGTACGATCGATCGATCGCTAGACGTATCG"
seq_2 = "actgatcgacgatcgatcgatcacgact"
seq_3 = "ACTGAC-ACTGT—ACTGTA----CATGTG"

# make three files to hold the output
output_1 = open(header_1 + ".fasta", "w")
output_2 = open(header_2 + ".fasta", "w")
output_3 = open(header_3 + ".fasta", "w")

# write one sequence to each output file
output_1.write('>' + header_1 + '\n' + seq_1 + '\n')
output_2.write('>' + header_2 + '\n' + seq_2.upper() + '\n')
output_3.write('>' + header_3 + '\n' + seq_3.replace('-', '') + '\n')
```

Looking at the code above, it seems like there's a lot of redundancy there. Each of the four sections of code – setting the header values, setting the sequence values, creating the output files, and writing data to the output files – consists of three nearly-identical statements. Although the solution works, it seems to involve a lot of unnecessary typing! Also, having so much nearly-identical code seems likely to cause errors if we need to change something. In the next chapter, we'll examine some tools which will allow us to start removing some of that redundancy.

4: Lists and loops

Why do we need lists and loops?

Think back over the exercises that we've seen in the previous two chapters – they've all involved dealing with one bit of information at a time. In chapter 2, we used string manipulation tools to process single sequences, and in chapter 3, we practised reading and writing files one at a time. The closest we got to using multiple pieces of data was during the final exercise in chapter 3, where we were dealing with three DNA sequences.

If that's all that Python allowed us to do, it wouldn't be a very helpful tool for biology. In fact, there's a good chance that you're reading this book because you want to be able to write programs to help you deal with large datasets. A very common situation in biological research is to have a large collection of data (DNA sequences, SNP positions, gene expression measurements) that all need to be processed in the same way. In this chapter, we'll learn about the fundamental programming tools that will allow our programs to do this.

So far we have learned about several different data types (strings, numbers, and file objects), all of which store a single bit of information[1]. When we've needed to store multiple bits of information (for example, the three DNA sequences in the chapter 3 exercises) we have simply created more variables to hold them:

```
# set the values of all the sequence variables
seq_1 = "ATCGTACGATCGATCGATCGCTAGACGTATCG"
seq_2 = "actgatcgacgatcgatcgatcacgact"
seq_3 = "ACTGAC-ACTGT–ACTGTA----CATGTG"
```

[1] We know that files are slightly different to strings and numbers because they can store a lot of information, but each file object still only refers to a single file.

The limitations of this approach became clear quite quickly as we looked at the solution code – it only worked because the number of sequences were small, and we knew the number in advance. If we were to repeat the exercise with three hundred or three thousand sequences, the vast majority of the code would be given over to storing variables and it would become completely unmanageable. And if we were to try and write a program that could process an unknown number of input sequences (for instance, by reading them from a file), we wouldn't be able to do it. To make our programs able to process multiple pieces of data, we need an entirely new type of structure which can hold many pieces of information at the same time – a *list*.

We've also dealt exclusively with programs whose statements are executed from top to bottom in a very straightforward way. This has great advantages when first starting to think about programming – it makes it very easy to follow the flow of a program. The downside of this sequential style of programming, however, is that it leads to very redundant code like we saw at the end of the previous chapter:

```
# make three files to hold the output
output_1 = open(header_1 + ".fasta", "w")
output_2 = open(header_2 + ".fasta", "w")
output_3 = open(header_3 + ".fasta", "w")
```

Again; it was only possible to solve the exercise in this manner because we knew in advance the number of output files we were going to need. Looking at the code, it's clear that these three lines consist of essentially the same statement being executed multiple times, with some slight variations. This idea of repetition-with-variation is incredibly common in programming problems, and Python has built in tools for expressing it – *loops*.

Creating lists and retrieving elements

To make a new list, we put several strings or numbers[1] inside square brackets, separated by commas:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
conserved_sites = [24, 56, 132]
```

Each individual item in a list is called an *element*. To get a single element from the list, write the variable name followed by the *index* of the element you want in square brackets:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
conserved_sites = [24, 56, 132]
print(apes[0])
first_site = conserved_sites[2]
```

If we want to go in the other direction – i.e. we know which element we want but we don't know the index – we can use the `index` method:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
chimp_index = apes.index("Pan troglodytes")
# chimp_index is now 1
```

Remember that in Python we start counting from zero rather than one, so the first element of a list is always at index zero. If we give a negative number, Python starts counting from the **end** of the list rather than the beginning – so it's easy to get the last element from a list:

```
last_ape = apes[-1]
```

[1] Or in fact, any other type of value or variable

What if we want to get more than one element from a list? We can give a start and stop position, separated by a colon, to specify a range of elements:

```
ranks = ["kingdom","phylum", "class", "order", "family"]
lower_ranks = ranks[2:5]
# lower ranks are class, order and family
```

Does this look familiar? It's the exact same notation that we used to get substrings back in chapter 2, and it works in exactly the same way – numbers are **inclusive** at the start and **exclusive** at the end. The fact that we use the same notation for strings and lists hints at a deeper relationship between the two types. In fact, what we were doing when extracting substrings in chapter 2 was **treating a string as though it were a list of characters**. This idea – that we can treat a variable as though it were a list when it's not – is a powerful one in Python and we'll come back to it later in this chapter.

Working with list elements

To add another element onto the end of an existing list, we can use the `append` method:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
apes.append("Pan paniscus")
```

`append` is an interesting method because it actually changes the variable on which it's used – in the above example, the `apes` list goes from having three elements to having four. We can get the length of a list by using the `len` function, just like we did for strings:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
print("There are " + str(len(apes)) + " apes")
apes.append("Pan paniscus")
print("Now there are " + str(len(apes)) + " apes")
```

The output shows that the number of elements in `apes` really has changed:

```
There are 3 apes
Now there are 4 apes
```

We can concatenate two lists just as we did with strings, by using the plus symbol:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
monkeys = ["Papio ursinus", "Macaca mulatta"]
primates = apes + monkeys

print(str(len(apes)) + " apes")
print(str(len(monkeys)) + " monkeys")
print(str(len(primates)) + " primates")
```

As we can see from the output, this doesn't change either of the two original lists – it makes a brand new list which contains elements from both:

```
3 apes
2 monkeys
5 primates
```

If we want to add elements from a list onto the end of an existing list, changing it in the process, we can use the `extend` method. `extend` behaves like `append` but takes a *list* as its argument rather than a single *element*.

Here are two more list methods that change the variable they're used on: `reverse` and `sort`. Both `reverse` and `sort` work by changing the order of the elements in the list. If we want to print out a list to see how this works, we need to used `str` (just as we did when printing out numbers):

```
ranks = ["kingdom","phylum", "class", "order", "family"]
print("at the start : " + str(ranks))
ranks.reverse()
print("after reversing : " + str(ranks))
ranks.sort()
print("after sorting : " + str(ranks))
```

If we take a look at the output, we can see how the order of the elements in the list is changed by these two methods:

```
at the start : ['kingdom', 'phylum', 'class', 'order', 'family']
after reversing : ['family', 'order', 'class', 'phylum', 'kingdom']
after sorting : ['class', 'family', 'kingdom', 'order', 'phylum']
```

By default, Python sorts strings in alphabetical order and numbers in ascending numerical order[1].

Writing a loop

Imagine we wanted to take our list of apes:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
```

and print out each element on a separate line, like this:

```
Homo sapiens is an ape
Pan troglodytes is an ape
Gorilla gorilla is an ape
```

[1] We can sort in other ways too, but that's beyond the scope of this book

One way to do it would be to just print each element separately:

```
print(apes[0] + " is an ape")
print(apes[1] + " is an ape")
print(apes[2] + " is an ape")
```

but this is very repetitive and relies on us knowing the number of elements in the list. What we need is a way to say something along the lines of "*for each element in the list of apes, print out the element, followed by the words ' is an ape'*". Python's loop syntax allows us to express those instructions like this:

```
for ape in apes:
   print(ape + " is an ape")
```

Let's take a moment to look at the different parts of this loop. We start by writing `for x in y`, where `y` is the name of the list we want to process and `x` is the name we want to use for the current element each time round the loop.

`x` is just a variable name (so it follows all the rules that we've already learned about variable names), but it behaves slightly differently to all the other variables we've seen so far. In all previous examples, we create a variable and store something in it, and then the value of that variable doesn't change unless we change it ourselves. In contrast, when we create a variable to be used in a loop, we don't set its value – the value of the variable will be automatically set to each element of the list in turn, and it will be different each time round the loop.

Importantly, the loop variable `x` only exists inside the loop – it gets created at the start of each loop iteration, and disappears at the end. This means that once the loop has finished running for the last time, that variable is gone forever. When a variable is restricted to a block of code like this, we call it the variable's *scope* – we will see several more examples later in the book.

This first line of the loop ends with a colon, and all the subsequent lines (just one, in this case) are indented. Indented lines can start with any number of tab or space characters, but they must all be indented in the same way. This pattern – a line which ends with a colon, followed by some indented lines – is very common in Python, and we'll see it in several more places throughout this book. A group of indented lines is often called a *block* of code[1].

In this case, we refer to the indented bock as the *body* of the loop, and the lines inside it will be executed once for each element in the list. To refer to the current element, we use the variable name that we wrote in the first line. The body of the loop can contain as many lines as we like, and can include all the functions and methods that we've learned about, with one important exception: we're not allowed to change the list while inside the body of the loop[2].

Here's an example of a loop with a more complicated body:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
for ape in apes:
    name_length = len(ape)
    first_letter = ape[0]
    print(ape + " is an ape. Its name starts with " + first_letter)
    print("Its name has " + str(name_length) + " letters")
```

The body of the loop in the code above has four statements, two of which are `print` statements, so each time round the loop we'll get two lines of output. If we look at the output we can see all six lines:

[1] If you're familiar with any other programming languages, you might know code blocks as things that are surrounded with curly brackets – the indentation does the same job in Python

[2] Changing the list while looping can cause Python to become confused about which elements have already been processed and which are yet to come.

```
Homo sapiens is an ape. Its name starts with H
Its name has 12 letters
Pan troglodytes is an ape. Its name starts with P
Its name has 15 letters
Gorilla gorilla is an ape. Its name starts with G
Its name has 15 letters
```

Why is the above approach better than printing out these six lines in six separate statements? Well, for one thing, there's much less redundancy – here we only needed to write two print statements. This also means that if we need to make a change to the code, we only have to make it once rather than three separate times. Another benefit of using a loop here is that if we want to add some elements to the list, we don't have to touch the loop code at all. Consequently, it doesn't matter how many elements are in the list, and it's not a problem if we don't know how many are going to be in it at the time when we write the code.

Indentation errors

Unfortunately, introducing tools like loops that require an indented block of code also introduces the possibility of a new type of error – an `IndentationError`. Notice what happens when the indentation of one of the lines in the block does not match the others:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
for ape in apes:
    name_length = len(ape)
  first_letter = ape[0]
    print(ape + " is an ape. Its name starts with " + first_letter)
    print("Its name has " + str(name_length) + " letters")
```

When we run this code, we get an error message before the program even starts to run:

```
IndentationError: unindent does not match any outer indentation level
```

When you encounter an `IndentationError`, go back to your code and double-check that all the lines in the block match up. Also double-check that you are using either tabs or spaces for indentation, **not both**. The easiest way to do this, as mentioned in chapter 1, is to enable *tab emulation* in your text editor.

Using a string as a list

We've already seen how a string can pretend to be a list – we can use list index notation to get individual characters or substrings from inside a string. Can we also use loop notation to process a string as though it were a list? Yes – if we write a loop statement with a string in the position where we'd normally find a list, Python treats **each character** in the string as a separate element. This allows us to very easily process a string one character at a time:

```
name = "martin"
for character in name:
    print("one character is " + character)
```

In this case, we're just printing each individual character:

```
one character is m
one character is a
one character is r
one character is t
one character is i
one character is n
```

The process of repeating a set of instructions for each element of a list (or character in a string) is called *iteration*, and we often talk about *iterating over* a list or string.

Splitting a string to make a list

So far in this chapter, all our lists have been written manually. However, there are plenty of functions and methods in Python that produce lists as their output. One such method that is particularly interesting to biologists is the `split` method which works on strings. `split` takes a single argument, called the *delimiter*, and splits the original string wherever it sees the delimiter, producing a list. Here's an example:

```
names = "melanogaster,simulans,yakuba,ananassae"
species = names.split(",")
print(str(species))
```

We can see from the output that the string has been split wherever there was a comma leaving us with a list of strings:

```
['melanogaster', 'simulans', 'yakuba', 'ananassae']
```

Of course, once we've created a list in this way we can iterate over it using a loop, just like any other list.

Iterating over lines in a file

Another very useful thing that we can iterate over is a file. Just as a string can pretend to be a list for the purposes of looping, a file object can do the same trick[1]. When we treat a string as a list, each character becomes an individual element, but when we treat a file as a list, each **line** becomes an individual element. This makes processing a file line-by-line very easy:

[1] If you're interested in how this "pretending" actually works, look up the Python documentation for *iterators* – but be prepared to do quite a bit of reading!

```
file = open("some_input.txt")
for line in file:
    # do something with the line
```

A quick warning: when you're writing a program that reads data from a file, it's best to use **either** the `read` method (to store the entire contents in a variable) **or** the loop method (to deal with each line separately). If you try to mix them, you might get unexpected behaviour. The reason for this is that Python keeps track of its position in each file, so if you read the contents of a file object using the `read` method, and then later try to process it one line at a time with a loop, you won't get any input because Python thinks it's already at the end of the file. If you absolutely have to use one method and then the other, you can get round this problem by closing and then re-opening the file.

Looping with ranges

Sometimes we want to loop over a list of numbers. Imagine we have a protein sequence:

```
protein = "vlspadktnv"
```

and we want to print out the first three residues, then the first four residues, etc. etc.:

```
vls
vlsp
vlspa
vlspad
...etc...
```

One way to tackle the problem would be to use a loop – we could extract a substring from the protein sequence and print it in the body of the loop, and the only thing that would need to change is the stop position in the substring. But what are we going to iterate over? We can't just iterate over the protein string, because that will give us individual residues, which is not what we want. We can manually assemble a list of stop positions, and loop over that:

```
stop_positions = [3,4,5,6,7,8,9,10]
for stop in stop_positions:
    substring = protein[0:stop]
    print(substring)
```

but this seems cumbersome, and only works if we know the length of the protein sequence in advance.

A better solution is to use the `range` function. `range` is a built-in Python function that generates lists of numbers for us to loop over. The behaviour of the range function depends on how many arguments we give it. Below are a few examples, with the output following directly after the code.

With a single argument, range will count up from zero to that number, excluding the number itself:

```
for number in range(6):
    print(number)
```

```
0
1
2
3
4
5
```

With two numbers, range will count up from the first number (inclusive[1]) to the second (exclusive):

```
for number in range(3, 8):
    print(number)
```

```
3
4
5
6
7
```

With three numbers, range will count up from the first to the second with the step size given by the third:

```
for number in range(2, 14, 4):
    print(number)
```

```
2
6
10
```

Recap

In this chapter we've seen several tools that work together to allow our programs to deal elegantly with multiple pieces of data. Lists let us store many elements in a single variable, and loops let us process those elements one by one. In learning about loops, we've also been introduced to the block syntax and the importance of indentation in Python.

[1] The rules for ranges are the same as for array notation – inclusive on the low end, exclusive on the high end – so you only have to memorize them once!

We've also seen several useful ways in which we can use the notation we've learned for working with lists with other types of data. Depending on the circumstances, we can use *strings*, *files*, and *ranges* as if they were lists. This is a very helpful feature of Python, because once we've become familiar with the syntax for working with lists, we can use it in many different place. Learning about these tools has also helped us make sense of some interesting behaviour that we observed in earlier chapters.

Exercises

Note: all the files mentioned in these exercises can be found in the *chapter_4* folder of the exercises download.

Processing DNA in a file

The file *input.txt* contains a number of DNA sequences, one per line. Each sequence starts with the same 14 base pair fragment – a sequencing adapter that should have been removed. Write a program that will (a) trim this adapter and write the cleaned sequences to a new file and (b) print the length of each sequence to the screen.

Multiple exons from genomic DNA

The file *genomic_dna.txt* contains a section of genomic DNA, and the file *exons.txt* contains a list of start/stop positions of exons. Each exon is on a separate line and the start and stop positions are separated by a comma. Write a program that will extract the exon segments, concatenate them, and write them to a new file.

Solutions

Processing DNA in a file

This seems a bit more complicated than previous exercises – we are being asked to write a program that does two things at once! – so lets tackle it one step at a time. First, we'll write a program that simply reads each sequence from the file and prints it to the screen:

```
file = open("input.txt")
for dna in file:
    print(dna)
```

We can see from the output that we've forgotten to remove the newlines from the ends of the DNA sequences – there is a blank line between each:

```
ATTCGATTATAAGCTCGATCGATCGATCGATCGATCGATCGATCGATCGATC

ATTCGATTATAAGCACTGATCGATCGATCGATCGATCGATGCTATCGTCGT

ATTCGATTATAAGCATCGATCACGATCTATCGTACGTATGCATATCGATATCGATCGTAGTC

ATTCGATTATAAGCACTATCGATGATCTAGCTACGATCGTAGCTGTA

ATTCGATTATAAGCACTAGCTAGTCTCGATGCATGATCAGCTTAGCTGATGATGCTATGCA
```

but we'll ignore that for now. The next step is to remove the first 14 bases of each sequence. We know that we want to take a substring from each sequence, starting at the fifteenth character, and continuing to the end. Unfortunately, the sequences are all different lengths, so the stop position is going to be different for all of them. We'll have to calculate the position of the last character for each sequence, by using the `len` function to calculate the length.

Here's what the code looks like with the substring part added:

```
file = open("input.txt")
for dna in file:
    last_character_position = len(dna)
    trimmed_dna = dna[14:last_character_position]
    print(trimmed_dna)
```

As before, we are simply printing the trimmed DNA sequence to the screen, and from the output we can confirm that the first 14 bases have been removed from each sequence:

```
TCGATCGATCGATCGATCGATCGATCGATCGATCGATCGATC

ACTGATCGATCGATCGATCGATCGATGCTATCGTCGT

ATCGATCACGATCTATCGTACGTATGCATATCGATATCGATCGTAGTC

ACTATCGATGATCTAGCTACGATCGTAGCTGTA

ACTAGCTAGTCTCGATGCATGATCAGCTTAGCTGATGATGCTATGCA
```

Now that we know our code is working, we'll switch from printing to the screen to writing to a file. We'll have to open the file **before** the loop, then write the trimmed sequences to the file **inside** the loop:

```
file = open("input.txt")
output = open("trimmed.txt", "w")
for dna in file:
    last_character_position = len(dna)
    trimmed_dna = dna[14:last_character_position]
    output.write(trimmed_dna)
```

Opening up the *trimmed.txt* file, we can see that the result looks good. It didn't matter that we never removed the newlines, because they appear in the correct place in the output file anyway:

```
TCGATCGATCGATCGATCGATCGATCGATCGATCGATCGATC
ACTGATCGATCGATCGATCGATCGATGCTATCGTCGT
ATCGATCACGATCTATCGTACGTATGCATATCGATATCGATCGTAGTC
ACTATCGATGATCTAGCTACGATCGTAGCTGTA
ACTAGCTAGTCTCGATGCATGATCAGCTTAGCTGATGATGCTATGCA
```

Now the final step – printing the lengths to the screen – requires just one more line of code. Here's the final program in full, with comments:

```
# open the input file
file = open("input.txt")

# open the output file
output = open("trimmed.txt", "w")

# go through the input file one line at a time
for dna in file:

    # calculate the position of the last character
    last_character_position = len(dna)

    # get the substring from the 15th character to the end
    trimmed_dna = dna[14:last_character_position]

    # print out the trimmed sequence
    output.write(trimmed_dna)

    # print out the length to the screen
    print("processed sequence with length " + str(len(trimmed_dna)))
```

Multiple exons from genomic DNA

This is very similar to the exercises from the previous two chapters, and so our solution to it is going to look very similar. Let's concentrate on the new bit of the problem first – reading the file of exon locations. As before, we can start by opening up the file and printing each line to the screen:

```
exon_locations = open("exons.txt")
for line in exon_locations:
    print(line)
```

This gives us a loop in which we are dealing with a different exon each time round. If we look at the output, we can see that we still have a newline at the end of each line, but we'll not worry about that for now:

```
5,58

72,133

190,276

340,398
```

Now we have to split up each line into a start and stop position. The `split` method is probably a good choice for this job – let's see what happens when we split each line using a comma as the delimiter:

```
exon_locations = open("exons.txt")
for line in exon_locations:
    positions = line.split(',')
    print(positions)
```

The output shows that each line, when split, turns into a list of two elements:

```
['5', '58\n']
['72', '133\n']
['190', '276\n']
['340', '398\n']
```

The second element of each list has a newline on the end, because we haven't removed them. Let's try assigning the start and stop position to sensible variable names, and printing them out individually:

```
exon_locations = open("exons.txt")
for line in exon_locations:
    positions = line.split(',')
    start = positions[0]
    stop = positions[1]
    print("start is " + start + ", stop is " + stop)
```

The output shows that this approach works – the start and stop variables take different values each time round the loop:

```
start is 5, stop is 58

start is 72, stop is 133

start is 190, stop is 276

start is 340, stop is 398

```

Now let's try putting these variables to use. We'll read the genomic sequence from the file all in one go using `read` – there's no need to process each line separately, as we just want the entire contents. Then we'll use the exon coordinates to extract one exon each time round the loop, and print it to the screen:

```
1 genomic_dna = open("genomic_dna.txt").read()
2 exon_locations = open("exons.txt")
3 for line in exon_locations:
4     positions = line.split(',')
5     start = positions[0]
6     stop = positions[1]
7     exon = genomic_dna[start:stop]
8     print("exon is: " + exon)
```

Unfortunately, when we run this code we get an error at line 7:

```
  File "multiple_exons_from_genomic_dna.py", line 7, in <module>
    exon = genomic_dna[start:stop]
TypeError: slice indices must be integers or None or have an __index__
method
```

What has gone wrong? Recall that the result of using `split` on a string is a list of strings – this means that the `start` and stop `variables` in our program are also strings (because they're just individual elements of the `positions` list). The problem comes when we try to use them as numbers in line 7. Fortunately, it's easily fixed – we just have to use the `int` function to turn our strings into numbers:

```
    start = int(positions[0])
    stop = int(positions[1])
```

and the program works as intended.

Next step: doing something useful with the exons, rather than just printing them to the screen. The exercise description says that we have to concatenate the exon sequences to make a long coding sequence. If we had all the exons in separate variables, then this would be easy;

```
coding_seq = exon1 + exon2 + exon3 + exon4
```

but instead we have a single `exon` variable that stores one exon at a time. Here's one way to get the complete coding sequence: before the loop starts we'll create a new variable called `coding_sequence` and assign it to an empty string. Then, each time round the loop, we'll add the current exon on to the end, and store the result back in the same variable. When the loop has finished, the variable will contain all the exons. This is what the code looks like (with line numbers as the program is getting quite long):

```
 1 genomic_dna = open("genomic_dna.txt").read()
 2 exon_locations = open("exons.txt")
 3 coding_sequence = ""
 4 for line in exon_locations:
 5     positions = line.split(',')
 6     start = int(positions[0])
 7     stop = int(positions[1])
 8     exon = genomic_dna[start:stop]
 9     coding_sequence = coding_sequence + exon
10     print("coding sequence is : " + coding_sequence)
```

On line 3 we create the `coding_sequence` variable, and on line 9, inside the loop, we add the current `exon` on to the end. This is an unusual type of variable assignment, because the `coding_sequence` variable is on both the left and right side of the equals sign. The trick to understanding line 9 is to read the right-hand side of the statement first i.e. "*concatenate the current* `coding_sequence` *and the current* `exon`*, then store the result of that concatenation in* `coding_sequence`".

On line 10, instead of printing the exon, we're printing the coding sequence, and we can see from the output how the coding sequence is gradually built up as we go round the loop:

```
coding sequence is : CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCG

coding sequence is :
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCGCGATCGATCGATATCGATCGA
TATCATCGATGCATCGATCATCGATCGATCGATCGATCGA
coding sequence is :
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCGCGATCGATCGATATCGATCGA
TATCATCGATGCATCGATCATCGATCGATCGATCGATCGACGATCGATCGATCGTAGCTAGCTAGCTAGATCGA
TCATCATCGTAGCTAGCTCGACTAGCTACGTACGATCGATGCATCGATCGTA
coding sequence is :
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCGCGATCGATCGATATCGATCGA
TATCATCGATGCATCGATCATCGATCGATCGATCGATCGACGATCGATCGATCGTAGCTAGCTAGCTAGATCGA
TCATCATCGTAGCTAGCTCGACTAGCTACGTACGATCGATGCATCGATCGTACGATCGATCGATCGATCGATCG
ATCGATCGATCGATCGATCGTAGCTAGCTACGATCG
```

The final step is to save the coding sequence to a file. We can do this at the end of the program with three lines of code. Here's the final code with comments:

```
# open the genomic dna file and read the contents
genomic_dna = open("genomic_dna.txt").read()

# open the exons locations file
exon_locations = open("exons.txt")

# create a variable to hold the coding sequence
coding_sequence = ""

# go through each line in the exon locations file
for line in exon_locations:

    # split the line using a comma
    positions = line.split(',')

    # get the start and stop positions
    start = int(positions[0])
    stop = int(positions[1])

    # extract the exon from the genomic dna
    exon = genomic_dna[start:stop]

    # append the exon to the end of the current coding sequence
    coding_sequence = coding_sequence + exon

# write the coding sequence to an output file
output = open("coding_sequence.txt", "w")
output.write(coding_sequence)
output.close()
```

5: Writing our own functions

Why do we want to write our own functions?

Take a look back at the very first exercise in this book – the one in chapter 2 where we had to write a program to calculate the AT content of a DNA sequence. Let's remind ourselves of the code:

```
my_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
length = len(my_dna)
a_count = my_dna.count('A')
t_count = my_dna.count('T')
at_content = (a_count + t_count) / length
print("AT content is " + str(at_content))
```

If we discount the first line (whose job is to store the input sequence) and the last line (whose job is to print the result), we can see that it takes four lines of code to calculate the AT content[1]. This means that every place in our code where we want to calculate the AT content of a sequence, we need these same four lines – and we have to make sure we copy them exactly, without any mistakes.

It would be much simpler if Python had a built-in function (let's call it `get_at_content`) for calculating AT content. If that were the case, then we could just run `get_at_content` in the same way we run `print`, or `len`, or `open`. Although, sadly, Python does not have such a built-in function, it does have the next best thing – a way for us to create our own functions.

Creating our own function to carry out a particular job has many benefits. It allows us to re-use the same code many times within a program without having to copy it out each time. Additionally, if we find that we have to make a change to the code,

[1] We could, of course, compress this down to a single line:
`at_content = (my_dna.count('A') + my_dna.count('T')) / len(my_dna)`
but it would be much less readable.

we only have to do it in one place. Splitting our code into functions also allows us to tackle larger problems, as we can work on different bits of the code independently. We can also re-use code across multiple programs.

Defining a function

Let's go ahead and create our `get_at_content` function. Before we start typing, we need to figure out what the inputs (the number and types of the *function arguments*) and outputs (the type of the *return value*) are going to be. For this function, that seems pretty obvious – the input is going to be a single DNA sequence, and the output is going to be a decimal number. To translate these into Python terms: the function will take a single argument of type *string*, and will return a value of type *number*[1]. Here's the code:

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your program:

```
from __future__ import division
```

The first line of the function definition contains a several different elements. We start with the word `def`, which is short for *define* (writing a function is called *defining* it). Following that we write the name of the function, followed by the names of the argument variables in parentheses. Just like we saw before with

[1] In fact, we can be a little bit more specific: we can say that the return value will be of type `float` – a floating point number (i.e. one with a decimal point).

normal variables, the function name and the argument names are arbitrary – we can use whatever we like.

The first line ends with a colon, just like the first line of the loops that we were looking at in the previous chapter. And just like loops, this line is followed by a *block* of indented lines – the *function body*. The function body can have as many lines of code as we like, as long as they all have the same indentation. Within the function body, we can refer to the arguments by using the variable names from the first line. In this case, the variable `dna` refers to the sequence that was passed in as the argument to the function.

The last line of the function causes it to return the AT content that was calculated in the function body. To `return` from a function, we simply write return followed by the value that the function should output.

There are a couple of important things to be aware of when writing functions. Firstly, we need to make a clear distinction between *defining* a function, and *running* it (we refer to running a function as *calling* it). The code we've written above will not cause anything to happen when we run it, because we've not actually asked Python to execute the `get_at_content` function – we have simply defined what it is. The code in the function will not be executed until we call the function like this:

```
get_at_content("ATGACTGGACCA")
```

If we simply call the function like that, however, then the AT content will vanish once it's been calculated. In order to use the function to do something useful, we must either store the result in a variable:

```
at_content = get_at_content("ATGACTGGACCA")
```

Or use it directly:

```
print("AT content is " + str(get_at_content("ATGACTGGACCA")))
```

Secondly, it's important to understand that the argument variable `dna` does not hold any particular value when the function is defined[1]. Instead, its job is to hold whatever value is given as the argument when the function is called. In this way it's analogous to the loop variables we saw in the previous chapter: loop variables hold a different value each time round the loop, and function argument variables hold a different value each time the function is called.

Finally, be aware that the same scoping rules that applied to loops also apply to functions – any variables that we create as part of the function only exist inside the function, and cannot be accessed outside. If we try to use a variable that's created inside the function from outside:

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

print(a_count)
```

We'll get an error:

```
NameError: name 'a_count' is not defined
```

[1] Indeed, it doesn't actually exist when it's defined, only when it runs.

Calling and improving our function

Let's write a small program that uses our new function, to see how it works. We'll try both storing the result in a variable before printing it (lines 8 and 9) and printing it directly (lines 10 and 11):

```
 1 def get_at_content(dna):
 2     length = len(dna)
 3     a_count = dna.count('A')
 4     t_count = dna.count('T')
 5     at_content = (a_count + t_count) / length
 6     return at_content
 7
 8 my_at_content = get_at_content("ATGCGCGATCGATCGAATCG")
 9 print(str(my_at_content))
10 print(get_at_content("ATGCATGCAACTGTAGC"))
11 print(get_at_content("aactgtagctagctagcagcgta"))
```

Looking at the output, we can see that the first function call works fine – the AT content is calculated to be 0.45, is stored in the variable my_at_content, then printed. However, the output for the next two calls is not so great. The call at line 10 produces a number with way too many figures after the decimal point, and the call at line 11, with the input sequence in lower case, gives a result of 0.0, which is definitely not correct:

```
0.45
0.5294117647058824
0.0
```

We'll fix these problems by making a couple of changes to the `get_at_content` function. We can add a rounding step in order to limit the number of significant figures in the result. Python has a built-in `round` function that takes two arguments – the number we want to round, and the number of significant figures. We'll call the `round` function on the result before we return it. And we can fix the lower case problem by converting the input sequence to upper case before starting the calculation. Here's the new version of the function, with the same three function calls:

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, 2)

my_at_content = get_at_content("ATGCGCGATCGATCGAATCG")
print(str(my_at_content))
print(get_at_content("ATGCATGCAACTGTAGC"))
print(get_at_content("aactgtagctagctagcagcgta"))
```

and now the output is just as we want:

```
0.45
0.53
0.52
```

We can make the function even better though: why not allow it to be called with the number of significant figures as an argument[1]? In the above code, we've picked two significant figures, but there might be situations where we want to see more. Adding the second argument is easy; we just add it to the argument variable list on the first line of the function definition, and then use the new argument variable in the call to `round`. We'll throw in a few calls to the new version of the function with different arguments to check that it works:

[1] An even better solution would be to specify the number of significant figures in the string representation of the number when it's printed.

```
def get_at_content(dna, sig_figs):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)

test_dna = "ATGCATGCAACTGTAGC"
print(get_at_content(test_dna, 1))
print(get_at_content(test_dna, 2))
print(get_at_content(test_dna, 3))
```

The output confirms that the rounding works as intended:

```
0.5
0.53
0.529
```

Encapsulation with functions

Let's pause for a moment and consider what happened in the previous section. We wrote a function, and then wrote some code that used that function. In the process of writing the code that used the function, we discovered a couple of problems with our original function definition. **We were then able to go back and change the function definition, without having to make any changes to the code that used the function**.

I've written that last sentence in bold, because it's incredibly important. It's no exaggeration to say that understanding the implications of that sentence is the key to being able to write larger, useful programs. The reason it's so important is that it describes a programming phenomenon that we call *encapsulation*. Encapsulation just means dividing up a complex program into little bits which we can work on independently. In the example above, the code is divided into two parts – the part where we define the function, and the part where we use it – and we can make changes to one part without worrying about the effects on the other part.

This is a very powerful idea, because without it, the size of programs we can write is limited to the number of lines of code we can hold in our head at one time. Some of the example code in the solutions to exercises in the previous chapter were starting to push at this limit already, even for relatively simple problems. By contrast, using functions allows us to build up a complex program from small building blocks, each of which individually is small enough to understand in its entirety.

Because using functions is so important, future solutions to exercises will use them when appropriate, even when it's not explicitly mentioned in the problem text. I encourage you to get into the habit of using functions in your solutions too.

Functions don't always have to take an argument

There's nothing in the rules of Python to say that your function **must** take an argument. It's perfectly possible to define a function with no arguments:

```
def get_a_number():
    return 42
```

but such functions tend not to be very useful. For example, we can write a version of `get_at_content` that doesn't require any arguments by setting the value of the `dna` variable inside the function:

```
def get_at_content():
    dna = "ACTGATGCTAGCTA"
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, 2)
```

but that's obviously not very useful. Occasionally you may be tempted to write a no-argument function that works like this:

```
1 def get_at_content():
2     length = len(dna)
3     a_count = dna.upper().count('A')
4     t_count = dna.upper().count('T')
5     at_content = (a_count + t_count) / length
6     return round(at_content, 2)
7
8 dna = "ACTGATCGATCG"
9 print(get_at_content())
```

At first this seems like a good idea – it works because the function gets the value of the dna variable that is set on line 8[1]. However, this breaks the encapsulation that we worked so hard to achieve. The function now only works if there is a variable called dna set in the bit of the code where the function is called, so the two pieces of code are no longer independent.

If you find yourself writing code like this, it's usually a good idea to identify which variables from outside the function are being used inside it, and turn them into arguments.

[1] It doesn't matter that the variable is set *after* the function is defined – all that matters it that it's set *before* the function is called on line 9.

Functions don't always have to return a value

Consider this variation of our function – instead of *returning* the AT content, this function *prints* it to the screen:

```
def print_at_content(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    print(str(round(at_content, 2)))
```

When you first start writing functions, it's very tempting to do this kind of thing. You think "*OK, I need to calculate and print the AT content – I'll write a function that does both*". The trouble with this approach is that it results in a function that is less flexible. Right now you want to print the AT content to the screen, but what if you later discover that you want to write it to a file, or use it as part of some other calculation? You'll have to write more functions to carry out these tasks.

The key to designing flexible functions is to recognize that the job *calculate and print the AT content* is actually two separate jobs – calculating the AT content, and printing it. Try to write your functions in such a way that they just do one job. You can then easily write code to carry out more complicated jobs by using your simple functions as building blocks.

Functions can be called with named arguments

What do we need to know about a function in order to be able to use it? We need to know what the return value and type is, and we need to know the number and type of the arguments. For the examples we've seen so far in this book, we also need to know the **order** of the arguments. For instance, to use the `open` function we need to know that the name of the file comes first, followed by the mode of the file. And to use our two-argument version of `get_at_content` as described above, we need to know that the DNA sequence comes first, followed by the number of significant figures.

There's a feature in Python called *keyword arguments* which allows us to call functions in a slightly different way. Instead of giving a list of arguments in parentheses:

```
get_at_content("ATCGTGACTCG", 2)
```

we can supply a list of argument variable names and values joined by equals signs:

```
get_at_content(dna="ATCGTGACTCG", sig_figs=2)
```

This style of calling functions[1] has several advantages. It doesn't rely on the order of arguments, so we can use whichever order we prefer. These two statements behave identically:

```
get_at_content(dna="ATCGTGACTCG", sig_figs=2)
get_at_content(sig_figs=2, dna="ATCGTGACTCG")
```

It's also clearer to read what's happening when the argument names are given explicitly.

We can even mix and match the two styles of calling – the following are all identical:

```
get_at_content("ATCGTGACTCG", 2)
get_at_content(dna="ATCGTGACTCG", sig_figs=2)
get_at_content("ATCGTGACTCG", sig_figs=2)
```

Although we're not allowed to start off with keyword arguments then switch back to normal – this will cause an error:

[1] It works with methods too, including all the ones we've seen so far.

```
get_at_content(dna="ATCGTGACTCG", 2)
```

Keyword arguments can be particularly useful for functions and methods that have a lot of arguments, and we'll use them where appropriate in the examples and exercise solutions in the rest of this book.

Function arguments can have defaults

We've encountered function arguments with defaults before, when we were discussing opening files in chapter 3. Recall that the `open` function takes two arguments – a file name and a mode string – but that if we call it with **just** a file name it uses a default value for the mode string. We can easily take advantage of this feature in our own functions – we simply specify the default value in the first line of the function definition. Here's a version of our `get_at_content` function where the default number of significant figures is two:

```
def get_at_content(dna, sig_figs=2):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)
```

Now we have the best of both worlds. If the function is called with two arguments, it will use the number of significant figures specified; if it's called with one argument, it will use the default value of two significant figures. Let's see some examples:

```
get_at_content("ATCGTGACTCG")
get_at_content("ATCGTGACTCG", 3)
get_at_content("ATCGTGACTCG", sig_figs=4)
```

The function takes care of filling in the default value for `sig_figs` for the first function call where none is supplied:

```
0.45
0.455
0.4545
```

Function argument defaults allow us to write very flexible functions which can have varying numbers of arguments. It only makes sense to use them for arguments where a sensible default can be chosen – there's no point specifying a default for the `dna` argument in our example. They are particularly useful for functions where some of the options are only going to be used infrequently.

Testing functions

When writing code of any type, it's important to periodically check that your code does what you intend it to do. If you look back over the solutions to exercises from the first few chapters, you can see that we generally test our code at each step by printing some output to the screen and checking that it looks OK. For example, in chapter 2 when we were first calculating AT content, we used a very short test sequence to verify that our code worked before running it on the real input.

The reason we used a test sequence was that, because it was so short, we could easily work out the answer by eye and compare it to the answer given by our code. This idea – running code on a test input and comparing the result to an answer **that we know to be correct**[1] – is such a useful one that Python has a built-in tool for expressing it: `assert`. An assertion consists of the word assert, followed by a call to our function, then **two** equals signs, then the result that we expect[2].

[1] Think of it as similar to running a positive control in a wet-lab experiment.

[2] In fact, assertions can include any conditional statement; we'll learn about those in the next chapter.

For example, we know that if we run our `get_at_content` function on the DNA sequence "ATGC" we should get an answer of 0.5. This assertion will test whether that's the case:

```
assert get_at_content("ATGC") == 0.5
```

Notice the two equals signs – we'll learn the reason behind that in the next chapter. The way that assertion statements work is very simple; if an assertion turns out to be false (i.e. if Python executes our function on the input "ATGC" and the answer isn't 0.5) then the program will stop and we will get an `AssertionError`.

Assertions are useful in a number of ways. They provide a means for us to check whether our functions are working as intended and therefore help us track down errors in our programs. If we get some unexpected output from a program that uses a particular function, and the assertion tests for that function all pass, then we can be confident that the error doesn't lie in the function but in the code that calls it.

They also let us modify a function and check that we haven't introduced any errors. If we have a function that passes a series of assertion tests, and we make some changes to it, we can re-run the assertion tests and, assuming they all pass, be confident that we haven't broken the function[1].

Assertions are also useful as a form of documentation. By including a collection of assertion tests alongside a function, we can show exactly what output is expected from a given input.

Finally, we can use assertions to test the behaviour of our function for unusual inputs. For example, what is the expected behaviour of `get_at_content` when given a DNA sequence that includes unknown bases (usually represented as `N`)? A sensible way to handle unknown bases would be to exclude them from the AT content calculation – in other words, the AT content for a given sequence shouldn't

[1] This idea is very similar to a process in software development called *regression testing*.

be affected by adding a bunch of unknown bases. We can write an assertion that expresses this:

```
assert get_at_content("ATGCNNNNNNNNNN") == 0.5
```

This assertions fails for the current version of `get_at_content`. However, we can easily modify the function to remove all `N` characters before carrying out the calculation:

```
def get_at_content(dna, sig_figs=2):
    dna = dna.replace('N', '')
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)
```

and now the assertion passes.

It's common to group a collection of assertions for a particular function together to test for the correct behaviour on different types of input. Here's an example for `get_at_content` which shows a range of different types of behaviour:

```
assert get_at_content("A") == 1
assert get_at_content("G") == 0
assert get_at_content("ATGC") == 0.5
assert get_at_content("AGG") == 0.33
assert get_at_content("AGG", 1) == 0.3
assert get_at_content("AGG", 5) == 0.33333
```

Recap

In this chapter, we've seen how packaging up code into functions helps us to manage the complexity of large programs and promote code reuse. We learned how to define and call our own functions along with various new ways to supply arguments to functions. We also looked at a couple of things that are possible in Python, but rarely advisable – writing functions without arguments or return values. Finally, we explored the use of assertions to test our functions, and discussed how we can use them to catch errors before they become a problem.

The remaining chapters in this book will make use of functions in both the examples and the exercise solutions, so make sure you are comfortable with the new ideas from this chapter before moving on.

Exercises

Percentage of amino acid residues, part one

Write a function that takes two arguments – a protein sequence and an amino acid residue code – and returns the percentage of the protein that the amino acid makes up. Use the following assertions to test your function:

```
assert my_function("MSRSLLLRFLLFLLLLPPLP", "M") == 5
assert my_function("MSRSLLLRFLLFLLLLPPLP", "r") == 10
assert my_function("MSRSLLLRFLLFLLLLPPLP", "L") == 50
assert my_function("MSRSLLLRFLLFLLLLPPLP", "Y") == 0
```

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your program:

```
from __future__ import division
```

Percentage of amino acid residues, part two

Modify the function from part one so that it accepts a list of amino acid residues rather than a single one. If no list is given, the function should return the percentage of hydrophobic amino acid residues (A, I, L, M, F, W, Y and V). Your function should pass the following assertions:

```
assert my_function("MSRSLLLRFLLFLLLLPPLP", ["M"]) == 5
assert my_function("MSRSLLLRFLLFLLLLPPLP", ['M', 'L']) == 55
assert my_function("MSRSLLLRFLLFLLLLPPLP", ['F', 'S', 'L']) == 70
assert my_function("MSRSLLLRFLLFLLLLPPLP") == 65
```

Solutions

Percentage of amino acid residues, part one

This is a similar problem to ones that we've tackled before, but we'll have to pay attention to the details. Let's start with a piece of code that does the calculation for a specific protein sequence and amino acid code, and then turn it into a function. Calculating the percentage is very similar to calculating the AT content, but we will need to multiple the result by 100 to get a percentage rather than a fraction:

```
protein = "MSRSLLLRFLLFLLLLPPLP"
aa = "R"
aa_count = protein.count(aa)
protein_length = len(protein)
percentage = aa_count * 100 / protein_length
print(percentage)
```

Now we'll make this code into a function by turning the two variables `protein` and `aa` into arguments, and returning the percentage rather than printing it. We'll add in the assertions at the end of the program to test if the function is doing its job:

```
def get_aa_percentage(protein, aa):
    aa_count = protein.count(aa)
    protein_length = len(protein)
    percentage = aa_count * 100 / protein_length
    return percentage

# test the function with assertions
assert get_aa_percentage("MSRSLLLRFLLFLLLLPPLP", "M") == 5
assert get_aa_percentage("MSRSLLLRFLLFLLLLPPLP", "r") == 10
assert get_aa_percentage("msrslllrfllfllllpplp", "L") == 50
assert get_aa_percentage("MSRSLLLRFLLFLLLLPPLP", "Y") == 0
```

Running the code shows that one of the assertions is failing – the error message tells us which assertion is the failed one:

```
    assert get_aa_percentage("MSRSLLLRFLLFLLLLPPLP", "r") == 10
AssertionError
```

Our function fails to work when the protein sequence is in upper case, but the amino acid residue code is in lower case. Looking at the assertions, we can make an educated guess that the next one (with the protein in lower case and the amino acid in upper case) is probably going to fail as well. Let's try to fix both of these problems by converting both the protein and the amino acid string to upper case at the start of the function. We'll use the same trick as we did before of converting a string to upper case and then storing the result back in the same variable:

```
def get_aa_percentage(protein, aa):

    # convert both inputs to upper case
    protein = protein.upper()
    aa = aa.upper()

    aa_count = protein.count(aa)
    protein_length = len(protein)
    percentage = aa_count * 100 / protein_length
    return percentage
```

Now all the assertions pass without error.

Percentage of amino acid residues, part two

This exercise involves something that we've not seen before: a function that takes a list as one of its arguments. As in the previous exercise, we'll pick one of the assertion cases and write the code to solve it first, then turn the code into a function.

There are actually two ways to approach this problem. We can use a loop to go through each of the given amino acid residues in turn, counting up the number of times they occur in the protein sequence, to get a total count. Or, we can treat the protein sequence string as a list (as described in the previous chapter) and ask, for each position, whether the character at that position is a member of the list of amino acid residues. We'll use the first method here; in the next chapter we'll learn about the tools necessary to implement the second.

We'll need some way to keep a running total of matching amino acids as we go round the loop, so we'll create a new variable outside the loop and update it each time round. The code inside the loop will be quite similar to that from the previous exercise. Here's the code with some `print` statements so we can see exactly what is happening:

```
protein = "MSRSLLLRFLLFLLLLPPLP"
aa_list = ['M', 'L', 'F']

# the total variable will hold the total number of matching residues
total = 0
for aa in aa_list:
    print("counting number of " + aa)
    aa = aa.upper()
    aa_count = protein.count(aa)

    # add the number for this residue to the total count
    total = total + aa_count
    print("running total is " + str(total))

percentage = total * 100 / len(protein)
print("final percentage is " + str(percentage))
```

When we run the code, we can see how the running total increases each time round the loop:

```
counting number of M
running total is 1
counting number of L
running total is 11
counting number of F
running total is 13
final percentage is 65.0
```

Now let's take the code and, just like before, turn the protein string and the amino acid list into arguments to create a function:

```
def get_aa_percentage(protein, aa_list):
    protein = protein.upper()
    protein_length = len(protein)
    total = 0
    for aa in aa_list:
        aa = aa.upper()
        aa_count = protein.count(aa)
        total = total + aa_count
    percentage = total * 100 / protein_length
    return percentage
```

This function passes all the assertion tests except the last one, which tests the behaviour when run with only one argument. In fact, Python never even gets as far as testing the result from running the function, as we get an error indicating that the function didn't complete:

```
TypeError: get_aa_percentage() takes exactly 2 arguments (1 given)
```

Fixing the error takes only one change: we add a default value for `aa_list` in the first line of the function definition:

```
def get_aa_percentage(protein, aa_list=['A','I','L','M','F','W','Y','V']):

    protein = protein.upper()
    protein_length = len(protein)
    total = 0
    for aa in aa_list:
        aa = aa.upper()
        aa_count = protein.count(aa)
        total = total + aa_count
    percentage = total * 100 / protein_length
    return percentage
```

and now all the assertions pass.

6: Conditional tests

Programs need to make decisions

If we look back at the examples and exercises in previous chapters, something that stands out is the lack of decision-making. We've gone from doing simple calculations on individual bits of data to carrying out more complicated procedures on collections of data, but the way that each bit of data (a sequence, a base, a species name, an exon) has been treated identically.

Real-life problems, however, often require our programs to act as decision-makers; to examine a property of some bit of data and decide what to do with it. In this chapter, we'll see how to do that using *conditional statements*. Conditional statements are features of Python that allow us to build decision points in our code. They allow our programs to decide which out of a number of possible courses of action to take – instructions like "*print the name of the sequence if it's longer than 300 bases*" or "*group two samples together if they were collected less than 10 metres apart*".

Before we can start using conditional statements, however, we need to understand *conditions*.

Conditions, True and False

A *condition* is simply a bit of code that can produce a true or false answer. The easiest way to understand how conditions work in Python is try out a few examples. The following example prints out the result of testing (or *evaluating*) a bunch of different conditions – some mathematical examples, some using string methods, and one for testing if a value is included in a list:

```
 1 print(3 == 5)
 2 print(3 > 5)
 3 print(3 <=5)
 4 print(len("ATGC") > 5)
 5 print("GAATTC".count("T") > 1)
 6 print("ATGCTT".startswith("ATG"))
 7 print("ATGCTT".endswith("TTT"))
 8 print("ATGCTT".isupper())
 9 print("ATGCTT".islower())
10 print("V" in ["V", "W", "L"])
```

If we look at the output, we can see use the line numbers to match up each condition with its result:

```
 1 False
 2 False
 3 True
 4 False
 5 True
 6 True
 7 False
 8 True
 9 False
10 True
```

But what's actually being printed here? At first glance, it looks like we're printing the strings "True" and "False", but those strings don't appear anywhere in our code. What is actually being printed is the special built-in values that Python uses to represent true and false – they are capitalized so that we know they're these special values.

We can show that these values are special by trying to print them. The following code runs without errors (note the absence of quotation marks):

```
print(True)
print(False)
```

whereas trying to print arbitrary unquoted words:

```
print(Hello)
```

causes a `NameError`.

There's a wide range of things that we can include in conditions, and it would be impossible to give an exhaustive list here. The basic building blocks are:

- equals (represented by ==)
- greater and less than (represented by `>` and `<`)
- greater and less than or equal to (represented by >= and <=)
- not equal (represented by `!=`)
- is a value in a list (represented by `in`)
- are two objects the same[1] (represented by `is`)

Many data types also provide methods that return True or False values, which are often a lot more convenient to use than the building blocks above. We've already seen a few in the code sample above: for example, strings have a `startswith` method that returns true if the string starts with the string given as an argument. We'll mention these true/false methods when they come up.

Notice that the test for equality is **two** equals signs, not one. Forgetting the second equals sign will cause an error.

Now that we know how to express tests as conditions, let's see what we can do with them.

[1] A discussion of what this actually means in Python is beyond the scope of this book, so we'll avoid using this comparison for the chapter.

if statements

The simplest kind of conditional statement is an if statement. Hopefully the syntax is fairly simple to understand:

```
expression_level = 125
if expression_level > 100:
    print("gene is highly expressed")
```

We write the word `if`, followed by a condition, and end the first line with a colon. There follows a block of indented lines of code (the *body* of the if statement), which will only be executed if the condition is true. This colon-plus-block pattern should be familiar to you from the chapters on loops and functions.

Most of the time, we want to use an if statement to test a property of some variable whose value we don't know at the time when we are writing the program. The example above is obviously useless, as the value of the `expression_level` variable is not going to change!

Here's a slightly more interesting example: we'll define a list of gene accession names and print out just the ones that start with "a":

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        print(accession)
```

Looking at the output allows us to check that this works as intended:

```
ab56
ay93
ap97
```

If you take a close look at the code above, you'll see something interesting – the lines of code inside the loop are indented (just as we've seen before), but the line of code inside the `if` statement is indented **twice** – once for the loop, and once for the `if`. This is the first time we've seen multiple levels of indentation, but it's very common once we start working with larger programs – whenever we have one loop or `if` statement nested inside another, we'll have this type of indentation.

Python is quite happy to have as many levels of indentation as needed, but you'll need to keep careful track of which lines of code belong at which level. If you find yourself writing a piece of code that requires more than three levels of indentation, it's generally an indication that that piece of code should be turned into a function.

else statements

Closely related to the `if` statement is the `else` statement. The examples above use a *yes/no* type of decision-making: should we print the gene accession number or not? Often we need an *either/or* type of decision, where we have two possible actions to take. To do this, we can add on an `else` clause after the end of the body of an `if` statement:

```
expression_level = 125
if expression_level > 100:
   print("gene is highly expressed")
else:
   print("gene is lowly expressed")
```

The `else` statement doesn't have any condition of its own – rather, the else statement body is execute when the if statement to which it's attached is **not** executed.

Here's an example which uses `if` and `else` to split up a list of accession names into two different files – accessions that start with "a" go into the first file, and all other accessions go into the second file:

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    else:
        file2.write(accession + "\n")
```

Notice how there are multiple indentation levels as before, but that the `if` and `else` statements are at the **same** level.

elif statements

What if we have more than two possible branches? For example, say we want three files of accession names: ones that start with "a", ones that start with "b", and all others. We could have a second `if` statement nested inside the `else` clause of the first `if` statement:

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
file3 = open("three.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    else:
        if accession.startswith('b'):
            file2.write(accession + "\n")
        else:
            file3.write(accession + "\n")
```

This works, but is difficult to read – we can quickly see that we need an extra level of indentation for every additional choice we want to include. To get round this, Python has an `elif` statement, which merges together `else` and `if` and allows us to rewrite the above example in a much more elegant way:

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
file3 = open("three.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    elif accession.startswith('b'):
        file2.write(accession + "\n")
    else:
        file3.write(accession + "\n")
```

Notice how this version of the code only needs two levels of indention. In fact, using `elif` we can have any number of branches and still only require a single extra level of indentation:

```
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    elif accession.startswith('b'):
        file2.write(accession + "\n")
    elif accession.startswith('c'):
        file3.write(accession + "\n")
    elif accession.startswith('d'):
        file4.write(accession + "\n")
    elif accession.startswith('e'):
        file5.write(accession + "\n")
    else:
        file6.write(accession + "\n")
```

while loops

Here's one final thing we can do with conditions: use them to determine when to exit a loop. In chapter 4 we learned about loops that *iterate over* a collection of items (like a list, a string or a file). Python has another type of loop called a `while` loop. Rather than running a set number of times, a `while` loop runs until some condition is met. For example, here's a bit of code that increments a count variable by one each time round the loop, stopping when the count variable reaches ten:

```
count = 0
while count<10:
    print(count)
    count = count + 1
```

Because normal loops in Python are so powerful[1], `while` loops are used much less frequently than in other languages, so we won't discuss them further.

Building up complex conditions

What if we wanted to express a condition that was made up of several parts? Imagine we want to go through our list of accessions and print out only the ones that start with "a" and end with "3". We could use two nested if statements:

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        if accession.endswith('3'):
            print(accession)
```

but this brings in an extra, unneeded level of indention. A better way is to join up the two condition with `and` to make a complex expression:

[1] E.g. the example code here could be better accomplished with a `range`.

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a') and accession.endswith('3'):
        print(accession)
```

This version is nicer in two ways: it doesn't require the extra level of indentation, and the condition reads in a very natural way. We can also use `or` to join up two conditions, to produce a complex condition that will be true if either of the two simple conditions are true:

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a') or accession.startswith('b'):
        print(accession)
```

We can even join up complex conditions to make more complex conditions – here's an example which prints accessions if they start with either "a" or "b" and end with "4":

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for acc in accs:
    if (acc.startswith('a') or acc.startswith('b')) and acc.endswith('4'):
        print(acc)
```

Notice how we have to include parentheses in the above example to avoid ambiguity. Finally, we can negate any type of condition by prefixing it with the word `not`. This example will print out accessions that start with "a" and **don't** end with 6:

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for acc in accs:
    if acc.startswith('a') and not acc.endswith('6'):
        print(acc)
```

By using a combination of `and`, `or` and `not` (along with parentheses where necessary) we can build up arbitrarily complex conditions.

These three words are collectively known as *boolean operators* and crop up in a lot of places. For example, if you wanted to search for information on using Python in biology, but didn't want to see pages that talked about biology of snakes, you might do a search for "*biology python -snake*". This is actually a complex condition just like the ones above – Google automatically adds *and* between words, and uses the hyphen to mean *not*. So you're asking for pages that mention python *and* biology but *not* snakes.

Writing true/false functions

Sometimes we want to write a function that can be used in a condition. This is very easy to do – we just make sure that our function always returns either True or False. Remember that True and False are built-in values in Python, so they can be passed around, stored in variables, and returned, just like numbers or strings.

Here's a function that determines whether or not a DNA sequence is AT-rich (we'll say that a sequence is AT-rich if it has an AT content of more than 0.65):

```
def is_at_rich(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    if at_content > 0.65:
        return True
    else:
        return False
```

We'll test this function on a few sequences to see if it works:

```
print(is_at_rich("ATTATCTACTA"))
print(is_at_rich("CGGCAGCGCT"))
```

The output shows that the function returns True or False just like the other conditions we've been looking at:

```
True
False
```

Therefore we can use our function in an `if` statement:

```
if is_at_rich(my_dna):
    # do something with the sequence
```

Because the last four lines of our function are devoted to evaluating a condition and returning True or False, we can write a slightly more compact version. In this example we evaluate the condition, and then return the result right away:

```
def is_at_rich(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return at_content > 0.65
```

This is a little more concise, and also easier to read once you're familiar with the idiom.

Recap

In this short chapter, we've dealt with two things: conditions, and the statements that use them.

We've seen how simple conditions can be joined together to make more complex ones, and how the concepts of truth and falsehood are built in to Python on a fundamental level. We've also seen how we can incorporate True and False in our own functions in a way that allows them to be used as part of conditions.

We've been introduced to four different tools that use conditions – `if`, `else`, `elif`, and `while` – in approximate order of usefulness. You'll probably find, in the programs that you write and in your solutions to the exercises in this book, that you use `if` and `else` very frequently, `elif` occasionally, and `while` almost never.

Exercises

In the *chapter_6* folder in the exercises download, you'll find a text file called *data.csv*, containing some made-up data for a number of genes. Each line contains the following fields for a single gene in this order: species name, sequence, gene name, expression level. The fields are separated by commas (hence the name of the file – csv stands for Comma Separated Values). Think of it as a representation of a table in a spreadsheet – each line is a row, and each field in a line is a column. All the exercises for this chapter use the data read from this file.

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your programs:

```
from __future__ import division
```

Several species

Print out the gene names for all genes belonging to *Drosophila melanogaster* or *Drosophila simulans*.

Length range

Print out the gene names for all genes between 90 and 110 bases long.

AT content

Print out the gene names for all genes whose AT content is less than 0.5 and whose expression level is greater than 200.

Complex condition

Print out the gene names for all genes whose name begins with "k" or "h" except those belonging to *Drosophila melanogaster*.

High low medium

For each gene, print out a message giving the gene name and saying whether its AT content is high (greater than 0.65), low (less than 0.45) or medium (between 0.45 and 0.65).

Solutions

Several species

These exercises are somewhat more complicated than previous ones, and they're going to require material from multiple different chapters to solve. The first problem is to deal with the format of the data file. Open it up in a text editor and take a look before continuing.

We know that we're going to have to open the file (chapter 3) and process the contents line-by-line (chapter 4). To deal with each line, we'll have to split it to make a list of columns (chapter 4), then apply the condition (this chapter) in order to figure out whether or not we should print it. Here's a program that will read each line from the file, split it using commas as the delimiter, then assign each of the four columns to a variable and print the gene name:

```
data = open("data.csv")
for line in data:
    columns = line.rstrip("\n").split(",")
    species = columns[0]
    sequence = columns[1]
    name = columns[2]
    expression = columns[3]
    print(name)
```

Notice that we use `rstrip` to remove the newline from the end of the current line before splitting it. We know the order of the fields in the line because they were mentioned in the exercise description, so we can easily assign them to the four variables. This program doesn't do anything useful, but we can check the output to confirm that it gets the names right:

```
kdy647
jdg766
kdy533
hdt739
hdu045
teg436
```

Now we can add in the condition. We want to print the name if the species is **either** *Drosophila melanogaster* **or** *Drosophila simulans*. If the species name is **neither** of those two, then we don't want to do anything. This is a *yes/no* type decision, so we need an `if` statement:

```
data = open("data.csv")
for line in data:
    columns = line.rstrip("\n").split(",")
    species = columns[0]
    sequence = columns[1]
    name = columns[2]
    expression = columns[3]
    if species == "Drosophila melanogaster" or species == "Drosophila
simulans":
        print(name)
```

The line containing the if statement is quite long, so it wraps around onto the next line on this page, but it's still just a single line in the program file. We can check the output we get:

```
kdy647
jdg766
kdy533
```

against the contents of the file, and confirm that the program is working.

Length range

We can re-use a large part of the code from the previous exercise to help solve this one. We have another complex condition: we only want to print names for genes whose length is between 90 and 110 bases – in other words, genes whose length is greater than 90 **and** less than 110. We'll have to calculate the length using the `len` function. Once we've done that the rest of the program is quite straightforward:

```
data = open("data.csv")
for line in data:
    columns = line.rstrip("\n").split(",")
    species = columns[0]
    sequence = columns[1]
    name = columns[2]
    expression = columns[3]
    if len(sequence) > 90 and len(sequence) < 110:
        print(name)
```

AT content

This exercise has a complex condition like the others, but it also requires us to do a bit more calculation – we need to be able to calculate the AT content of each sequence. Rather than starting from scratch, we'll simply use the function that we wrote in the previous chapter and include it at the start of the program. Once we've done that, it's just a case of using the output from `get_at_content` as part of the condition:

```
# our function to get AT content
def get_at_content(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return at_content

data = open("data.csv")
for line in data:
    columns = line.rstrip("\n").split(",")
    species = columns[0]
    sequence = columns[1]
    name = columns[2]
    expression = columns[3]
    if get_at_content(sequence) < 0.5 and expression > 200:
        print(name)
```

Complex condition

There are no calculations to carry out for this exercise – the complexity comes from the fact that there are three components to the condition, and they have to be joined together in the right way:

```
data = open("data.csv")
for line in data:
    columns = line.rstrip("\n").split(",")
    species = columns[0]
    sequence = columns[1]
    name = columns[2]
    expression = columns[3]
    if (name.startswith('k') or name.startswith('h')) and species != 
"Drosophila melanogaster":
        print(name)
```

The line containing the if statement is quite long, so it wraps around onto the next line on this page, but it's still just a single line in the program file. There are two different ways to express the requirement that the name is not *Drosophila melanogaster*. In the above example we've used the not-equals sign (!=) but we could also have used the `not` boolean operator:

```
if (name.startswith('k') or name.startswith('h')) and not species ==
    "Drosophila melanogaster":
```

High low medium

Now we come to an exercise that requires the use of multiple branches. We have three different printing options for each gene – high, low and medium – so we'll need an `if..elif..else` section to handle the conditions. We'll use the `get_at_content` function as before:

```
# our function to get AT content
def get_at_content(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return at_content

data = open("data.csv")
for line in data:
    columns = line.rstrip("\n").split(",")
    species = columns[0]
    sequence = columns[1]
    name = columns[2]
    expression = columns[3]
    if get_at_content(sequence) > 0.65:
        print(name + " has high AT content")
    elif get_at_content(sequence) < 0.45:
        print(name + " has low AT content")
    else:
        print(name + " has medium AT content")
```

Checking the output confirms that the conditions are working:

```
kdy647 has high AT content
jdg766 has medium AT content
kdy533 has medium AT content
hdt739 has low AT content
hdu045 has medium AT content
teg436 has medium AT content
```

7: Regular expressions

The importance of patterns in biology

A lot of what we do when writing programs for biology can be described as searching for *patterns* in *strings*. The obvious examples come from the analysis of biological sequence data – remember that DNA, RNA and protein sequences are just strings. Many of the things we want to look for in biological sequences can be described in terms of patterns:

- protein domains
- DNA transcription factor binding motifs
- restriction enzyme cut sites
- degenerate PCR primer sites
- runs of mononucleotides

However, it's not just sequence data that can have interesting patterns. As we discussed in chapter 3, most of the other types of data we have to deal with in biology comes in the form of strings[1] inside text files – things like:

- read mapping locations
- geographical sample coordinates
- taxonomic names
- gene names
- gene accession numbers
- BLAST searches

[1] Note that although many of the things in this list are numerical data, they're still read in to Python programs as strings and need to be manipulated as such.

In previous chapters, we've looked at some programming tasks that involve pattern recognition in strings. We've seen how to count individual amino acid residues (and even groups of amino acid residues) in protein sequences (chapter 5), and how to identify restriction enzyme cut sites in DNA sequences (chapter 2). We've also seen how to examine parts of gene names and match them against individual characters (chapter 6).

The common theme among all these problems is that they involve searching for a **fixed** set of characters. But there are many problems that we want to solve that require more flexible patterns. For example:

- Given a DNA sequence, what's the length of the poly-A tail?
- Given a gene accession name, extract the part between the third character and the underscore
- Given a protein sequence, determine if it contains this highly-redundant domain motif

Because these types of problems crop up in so many different fields, there's a standard set of tools in Python[1] for dealing with them: *regular expressions*. Regular expressions[2] are a topic that might not be covered in a general-purpose programming book, but because they're so useful in biology, we're going to devote the whole of this chapter to looking at them.

Although the tools for dealing with regular expressions are built in to Python, they are not made automatically available when you write a program. In order to use them we must first talk about modules.

[1] And in many other languages and utilities.

[2] The name is often abbreviated to *regex*.

Modules in Python

The functions and data types that we've discussed so far in this book have been ones that are likely to be needed in pretty much every program – tools for dealing with strings and numbers, for reading and writing files, and for manipulating lists of data. As such, they are automatically made available when we start to create a Python program. If we want to open a file, we simply write a statement that uses the `open` function.

However, there's another category of tools in Python which are more specialized. Regular expressions are one example, but there is a large list of specialized tools which are very useful when you need them[1], but are not likely to be needed for the majority of programs. Examples include tools for doing advanced mathematical calculations, for downloading data from the web, for running external programs, and for manipulating date/time information. Each collection of specialized tools – really just a collection of specialized *functions* and data *types* – is called a *module*.

For reasons of efficiency, Python doesn't automatically make these modules available in each new program, as it does with the more basic tools. Instead, we have to explicitly load each module of specialized tools that we want to use inside our program. To load a module we use the `import` statement[2]. For example, the module that deals with regular expressions is called `re`, so if we want to write a program that uses the regular expression tools we must include the line:

```
import re
```

at the top of our program. When we then want to use one of the tools from a module, we have to prefix it with the module name[3]. For example, to use the

[1] Indeed, this is one of the great strengths of the Python language.

[2] This is the reason for the `from __future__ import division` statement that we have to include if we're using Python 2.

[3] There are ways round this, but we won't consider them in this book.

regular expression `search` function (which we'll discuss later in this chapter) we have to write:

```
re.search(pattern, string)
```

rather than simply:

```
search(pattern, string)
```

If we forget to import the module which we want to use, or forget to include the module name as part of the function call, we will get a `NameError`.

We'll encounter various other module in the rest of this book. For the rest of this chapter specifically, all code examples will require the `import re` statement in order to work. For clarity, we won't include it, so if you want try running any of the code in this chapter, you'll need to add it at the top.

Raw strings

Writing regular expression patterns, as we'll see in the very next section of this chapter, requires us to type a lot of special characters. Recall from chapter 2 that certain combinations of characters are interpreted by Python to have special meaning. For example, `\n` means *start a new line*, and `\t` means *insert a tab character*.

Unfortunately, there are a limited number of special characters to go round, so some of the characters that have a special meaning in regular expressions clash with the characters that **already** have a special meaning. Python's way round this problem is to have a special rule for strings: if we put the letter `r` immediately before the opening quotation mark, then any special characters inside the string are ignored:

```
print(r"\t\n")
```

The r stands for *raw*, which is Python's description for a string where special characters are ignored. Notice that the `r` goes **outside** the quotation marks – it is not part of the string itself. We can see from the output that the above code prints out the string just as we've written it:

```
\t\n
```

without any tabs or new lines. You'll see this special *raw* notation used in all the regular expression code examples in this chapter.

Searching for a pattern in a string

We'll start off with the simplest regular expression tool. `re.search` is a true/false function that determines whether or not a pattern appears somewhere in a string. It takes two arguments, both strings. The first argument is the pattern that you want to search for, and the second argument is the string that you want to search in. For example, here's how we test if a DNA sequence contains an EcoRI restriction site:

```
dna = "ATCGCGAATTCAC"
if re.search(r"GAATTC", dna):
   print("restriction site found!")
```

Notice that we've used the raw notation for the pattern, even though it's not strictly necessary as it doesn't contain any special characters.

Alternation

The above example isn't particularly interesting, as the restriction motif has no variation. Let's try it with the AvaII motif, which cuts at two different motifs:

GGACC and GGTCC. We can use the techniques we learned in the previous chapter to make a complex condition using `or`:

```
dna = "ATCGCGAATTCAC"
if re.search(r"GGACC", dna) or re.search(r"GGTCC", dna):
    print("restriction site found!")
```

But a better way is to capture the variation in the AvaII site using a regular expression:

```
dna = "ATCGCGAATTCAC"
if re.search(r"GG(A|T)CC", dna):
    print("restriction site found!")
```

Here we're using the alternation feature of regular expressions. Inside parentheses, we write the alternatives separated by a pipe character, so `(A|T)` means *either A or T*. This lets us write a single pattern – `GG(A|T)CC` – which captures the variation in the motif.

Character groups

The BisI restriction enzyme cuts at an even wider range of motifs – the pattern is GCNGC, where N represents any base. We can use the same alternation technique to search for this pattern:

```
dna = "ATCGCGAATTCAC"
if re.search(r"GC(A|T|G|C)GC", dna):
    print("restriction site found!")
```

However, there's another regular expression feature that lets us write the pattern more concisely. A pair of square brackets with a list of characters inside them can represent any one of these characters. So the pattern `GC[ATGC]GC` will match

`GCAGC`, `GCTGC`, `GCGGC` and `GCCGC`. Here's the same program using character groups:

```
dna = "ATCGCGAATTCAC"
if re.search(r"GC[ATGC]GC", dna):
   print("restriction site found!")
```

If we want a character in a pattern to match **any** character in the input, we can use a period – the pattern `GC.GC` would match all four possibilities. However, the period would also match any character which is not a DNA base, or even a letter. Therefore, the whole pattern would also match `GCFGC`, `GC&GC` and `GC9GC`, which may not be what we want.

Sometimes it's easier, rather than listing all the acceptable characters, to specify the characters that we **don't** want to match. Putting a caret ^ at the start of a character group like this [^XYZ] will negate it, and match any character that **isn't** in the group.

Quantifiers

The regular expression features discussed above let us describe variation in the individual characters of patterns. Another group of features, *quantifiers*, let us describe variation in the number of times a section of a pattern is repeated.

A question mark immediately following a character means that that character is optional – it can match either **zero or one times**. So in the pattern `GAT?C` the `T` is optional, and the pattern will match either `GATC` or `GAC`. If we want to apply a question mark to more than one character, we can group the characters in parentheses. For example, in the pattern GGG(AAA)?TTT the group of three `A`s is optional, so the pattern will match either `GGGAAATTT` or `GGGTTT`.

A plus sign immediately following a character or group means that the character or group **must** be present but can be repeated any number of times – in other words, it will match **one or more times**. For example, the pattern `GGGA+TTT` will match three `G`s, followed by one or more `A`s, followed by three `T`s. So it will match `GGGATTT`, `GGGAATT`, `GGGAAATT`, etc. but **not** `GGGTTT`.

An asterisk immediately following a character or group means that the character or group is optional, but can also be repeated. In other words, it will match **zero or more times**. For example, the pattern `GGGA*TTT` will match three `G`s, followed by zero or more `A`s, followed by three `T`s. So it will match `GGGTTT`, `GGGATTT`, `GGGAATTT`, etc.

If we want to specify a specific number of repeats, we can use curly brackets. Following a character or group with a **single** number inside curly brackets will match exactly that number of repeats. For example, the pattern `GA{5}T` will match `GAAAAAT` but not `GAAAAT` or `GAAAAAAT`. Following a character or group with a **pair of numbers** inside curly brackets separated with a comma allows us to specify an acceptable range of number of repeats. For example, the pattern `GA{2,4}T` will match `GAAT`, `GAAAT` and `GAAAAT` but not `GAT` or `GAAAAAT`.

Positions

The final set of regular expression tools we're going to look at don't represent characters at all, but rather positions in the input string. The caret symbol `^` matches the **start** of a string, and the dollar symbol `$` matches the **end** of a string. The pattern `^AAA` will match `AAATTT` but not `GGGAAATTT`. The pattern `GGG$` will match `AAAGGG` but not `AAAGGGCCC`.

Combining

The real power of regular expressions comes from combining these tools. We can use quantifiers together with alternations and character groups to specify very flexible patterns. For example, here's a complex pattern to identify full-length eukaryotic messenger RNA sequences:

```
^ATG[ATGC]{30,1000}A{5,10}$
```

Reading the pattern from left to right, it will match:

- an ATG start codon at the beginning of the sequence
- followed by between 30 and 1000 bases which can be A, T, G or C
- followed by a poly-A tail of between 5 and 10 bases at the end of the sequence

As you can see, regular expressions can be quite tricky to read until you're familiar with them! However, it's well worth investing a bit of time learning to use them, as the same notation is used across multiple different tools. The regular expression skills that you learn in Python are transferable to other programming languages, command line tools, and text editors.

The features we've discussed above are the ones most useful in biology, and are sufficient to tackle all the exercises at the end of the chapter. However, there are many more regular expression features available in Python. If you want to become a regular expression master, it's worth reading up on *greedy vs. minimal quantifiers*, *back-references*, *lookahead* and *lookbehind assertions*, and *built-in character classes*.

Before we move on to look at some more sophisticated uses of regular expressions, it's worth noting that there's a method similar to `re.search` called `re.match`. The difference is that `re.search` will identify a pattern occurring **anywhere** in the string, whereas `re.match` will only identify a pattern if it matches the **entire** string. Most of the time we want the former behaviour.

Extracting the part of the string that matched

In the section above we used `re.search` as the condition in an `if` statement to decide whether or not a string contained a pattern. Often in our programs, we want to find out not only **if** a pattern matched, but **what part** of the string was matched. To do this, we need to store the result of using `re.search`, then use the `group` method on the resulting object.

When introducing the `re.search` function above I wrote that it was a true/false function. That's not *exactly* correct though – if it finds a match, it doesn't return `True`, but rather an object that is evaluated as true in a conditional context[1] (if the distinction doesn't seem important to you, then you can safely ignore it). The value that's actually returned is a match object – a new data type that we've not encountered before. Like a file object (see chapter 3), a match object doesn't represent a simple thing, like a number or string. Instead, it represents the results of a regular expression search. And again, just like a file object, a match object has a number of useful methods for getting data out of it.

One such method is the `group` method. If we call this method on the result of a regular expression search, we get the portion of the input string that matched the pattern:

```
dna = "ATGACGTACGTACGACTG"

# store the match object in the variable m
m = re.search(r"GA[ATGC]{3}AC", dna)
print(m.group())
```

In the above code, we're searching inside a DNA sequence for `GA`, followed by three bases, followed by `AC`. By calling the `group` method on the resulting match object,

[1] If a match isn't found, then the same thing applies; the function doesn't return `False`, but a different built-in value – `None` – that evaluates as false. If this doesn't make sense, don't worry.

we can see the part of the DNA sequence that matched, and figure out what the middle three bases were:

```
GACGTAC
```

What if we want to extract more than one bit of the pattern? Say we want to match this pattern:

```
 GA[ATGC]{3}AC[ATGC]{2}AC
```

That's `GA`, followed by three bases, followed by `AC`, followed by two bases, followed by `AC` again. We can surround the bits of the pattern that we want to extract with parentheses – this is called *capturing* it:

```
 GA([ATGC]{3})AC([ATGC]{2})AC
```

We can now refer to the captured bits of the pattern by supplying an argument to the `group` method. `group(1)` will return the bit of the string matched by the section of the pattern in the first set of parentheses, `group(2)` will return the bit matched by the second, etc.:

```
dna = "ATGACGTACGTACGACTG"

# store the match object in the variable m
m = re.search(r"GA([ATGC]{3})AC([ATGC]{2})AC", dna)
print("entire match: " + m.group())
print("first bit: " + m.group(1))
print("second bit: " + m.group(2))
```

The output shows that the three bases in the first variable section were `CGT`, and the two bases in the second variable section were `GT`:

```
entire match: GACGTACGTAC
first bit: CGT
second bit: GT
```

Getting the position of a match

As well as containing information about the **contents** of a match, the match object also holds information about the **position** of the match. The `start` and `end` methods get the positions of the start and end of the pattern on the sequence:

```
dna = "ATGACGTACGTACGACTG"
m = re.search(r"GA([ATGC]{3})AC([ATGC]{2})AC", dna)
print("start: " + str(m.start()))
print("end: " + str(m.end()))
```

Remember that we start counting from zero, so in this case, the match starting at the third base has a start position of two:

```
start: 2
end: 13
```

We can get the start and end positions of individual groups by supplying a number as the argument to `start` and `end`:

```
dna = "ATGACGTACGTACGACTG"
m = re.search(r"GA([ATGC]{3})AC([ATGC]{2})AC", dna)
print("start: " + str(m.start()))
print("end: " + str(m.end()))
print("group one start: " + str(m.start(1)))
print("group one end: " + str(m.end(1)))
print("group two start: " + str(m.start(2)))
print("group two end: " + str(m.end(2)))
```

In this particular case, we could figure out the start and end positions of the individual groups from the start and end positions of the whole pattern:

```
start: 2
end: 13
group one start: 4
group one end: 7
group two start: 9
group two end: 11
```

but that might not always be possible for patterns that have variable length repeats.

Splitting a string using a regular expression

Occasionally it can be useful to split a string using a regular expression pattern as the delimiter. The normal string `split` method doesn't allow this, but the `re` module has a `split` function of its own that takes a regular expression pattern as an argument. The first argument is the pattern, the second argument is the string to be split.

Imagine we have a consensus DNA sequence that contains ambiguity codes, and we want to extract all runs of contiguous unambiguous bases. We need to split the DNA string wherever we see a base that isn't A, T, G or C:

```
dna = "ACTNGCATRGCTACGTYACGATSCGAWTCG"
runs = re.split(r"[^ATGC]", dna)
print(runs)
```

Recall that putting a caret ^ at the start of a character group negates it. The output shows how the function works – the return value is a list of strings:

```
['ACT', 'GCAT', 'GCTACGT', 'ACGAT', 'CGA', 'TCG']
```

Finding multiple matches

The examples we've seen so far deal with cases where we're only interested in a single occurrence of a pattern in a string. If instead we want to find every place where a pattern occurs in a string, there are two functions in the `re` module to help us.

`re.findall` returns a list of all matches of a pattern in a string. The first argument is the pattern, and the second argument is the string. Say we want to find all runs of `A` and `T` in a DNA sequence longer than five bases:

```
dna = "ACTGCATTATATCGTACGAAATTATACGCGCG"
runs = re.findall(r"[AT]{4,100}", dna)
print(runs)
```

Notice that the return value of the `findall` method is not a match object – it is a straightforward list of strings:

```
['ATTATAT', 'AAATTATA']
```

so we have no way to extract the positions. If we want to do anything more complicated than simply extracting the text of the matches, we need to use the `re.finditer` method. `finditer` returns a sequence of match objects, so to do anything useful with it, we need to use the return value in a loop:

```
dna = "ACTGCATTATATCGTACGAAATTATACGCGCG"
runs = re.finditer(r"[AT]{3,100}", dna)
for match in runs:
    run_start = match.start()
    run_end = match.end()
    print("AT rich region from " + str(run_start) + " to " + str(run_end))
```

As we can see from the output:

```
AT rich region from 5 to 12
AT rich region from 18 to 26
```

`finditer` gives us considerably more flexibility that `findall`.

Recap

Just as in the previous chapter, we learned about two distinct concepts (conditions, and the statements that use them) in this chapter we learned about *regular expressions*, and the *functions* that use them.

We started with a brief introduction to two concepts that, while not part of the regular expression tools, are necessary in order to use them – libraries and raw strings. We got a far-from-complete overview of features that can be used in regular expression patterns, and a quick look at the range of different things we can do with them. Just as regular expressions themselves can range from simple to complex, so can their uses. We can use regular expressions for simple tasks like determining whether or not a sequence contains a particular motif, or for complicated ones like identifying messenger RNA sequences by using complex patterns.

Before we move on to the exercises, it's important to recognize that for any given pattern, there are probably multiple ways to describe it using a regular expression. Near the start of the chapter, we came up with the pattern `GG(A|T)CC` to describe the AvaII restriction enzyme recognition site, but it could also be written as

- `GG[AT]CC`,
- `(GGACC|GGTCC)`
- `(GGA|GGT)CC`
- `G{2}[AT]C{2}`

As with other situations where there are multiple different ways to write the same thing, it's best to be guided by what is clearest to read.

Exercises

Accession names

Here's a list of made-up gene accession names:

xkn59438, yhdck2, eihd39d9, chdsye847, hedle3455, xjhd53e, 45da, de37dp

Write a program that will print only the accession names that satisfy the following criteria – treat each criterion separately:

- contain the number 5
- contain the letter d or e
- contain the letters d and e in that order
- contain the letters d and e in that order with a single letter between them
- contain both the letters d and e in any order
- start with x or y
- start with x or y and end with e
- contain three or more numbers in a row
- end with d followed by either a, r or p

Double digest

In the *chapter_7* file inside the exercises download, there's a file called *dna.txt* which contains a made-up DNA sequence. Predict the fragment lengths that we will get if we digest the sequence with two made-up restriction enzymes – AbcI, whose recognition site is ANT*AAT, and AbcII, whose recognition site is GCRW*TG (asterisks indicate the position of the cut site).

Solutions

Accession names

Obviously, the bulk of the work here is going to be coming up with the regular expression patterns to select each subset of the accession names. Here's the easy bit – storing the accession names in a list and then processing them in a loop (the first line wraps round because it's too long to fit on the page):

```
accs = ["xkn59438", "yhdck2", "eihd39d9", "chdsye847", "hedle3455",
"xjhd53e", "45da", "de37dp"]
for acc in accs:
   # print if it passes the test
```

Now we can tackle the different criteria one by one. For each example, the code (bordered by solid lines) is followed immediately by the output (bordered by dotted lines).

The first criterion is straightforward – accessions that contain the number 5. We don't even have to use any fancy regular expression features:

```
for acc in accs:
    if re.search(r"5", acc):
        print("\t" + acc)
```

```
   xkn59438
   hedle3455
   xjhd53e
   45da
```

Now for accessions that contain the letters d or e. We can use either alternation or a character group. Here's a solution using alternation:

```
for acc in accs:
    if re.search(r"(d|e)", acc):
        print("\t" + acc)
```

```
    yhdck2
    eihd39d9
    chdsye847
    hedle3455
    xjhd53e
    45da
    de37dp
```

The next one – accessions that contain both the letters d and e, in that order – is a bit more tricky. We can't just use a simple alternation or a character group, because they match **any** of their constituent parts, and we need **both** d and e. One way to think of the pattern is d, followed by some other letters and numbers, followed by e. We have to be careful with our quantifiers, however – at first glance the pattern `d.+e` looks good, but it will fail to match the accession where e follows d directly. To allow for the fact that d might be immediately followed by e, we need to use the asterisk:

```
for acc in accs:
    if re.search(r"d.*e", acc):
        print("\t" + acc)
```

```
    chdsye847
    hedle3455
    xjhd53e
    de37dp
```

The next requirement – d, followed by a single letter, followed by e – is actually easier to write a pattern for, even though it sounds more complicated. We simply remove the asterisk, and the period will now match any single character:

```
for acc in accs:
    if re.search(r"(d.e)", acc):
        print("\t" + acc)
```

```
    hedle3455
```

The next requirement – d and e in any order – is more difficult. We could do it with an alternation using the pattern `(d.*e|e.*d)`, which translates as *d then e, or e then d*. In this case, I think it's clearer to carry out two separate regular expression searches and combine them into a complex condition:

```
 for acc in accs:
    if re.search(r"d.*e", acc) or re.search(r"e.*d", acc):
        print("\t" + acc)
```

```
    hedle3455
    de37dp
```

To find accessions that start with either x or y, we need to combine an alternation with a start-of-string anchor:

```
 for acc in accs:
    if re.search(r"^(x|y)", acc):
        print("\t" + acc)
```

```
    xkn59438
    yhdck2
    xjhd53e
```

We can modify this quite easily to add the requirement that the accession ends with e. As before, we need to use .* in the middle to match any number of any character, resulting in quite a complex pattern:

```
  for acc in accs:
    if re.search(r"^(x|y).*e$", acc):
        print("\t" + acc)
```

```
    xjhd53e
```

To match three or more numbers in a row, we need a more specific quantifier – the curly brackets – and a character group which contains all the numbers:

```
for acc in accs:
    if re.search(r"[0123456789]{3,100}", acc):
        print("\t" + acc)
```

```
    xkn59438
    chdsye847
    hedle3455
```

We can actually make this a bit more concise. The character group of all digits is such a common one that there's a built-in shorthand for it: `\d`. We can also take advantage of a shorthand in the curly bracket quantifier – if we leave off the upper bound, then it matches with no upper limit. The more concise version:

```
for acc in accs:
    if re.search(r"\d{3,}", acc):
        print("\t" + acc)
    xkn59438
    chdsye847
    hedle3455
```

The final requirement is quite simple and only requires a character group and an end-of-string anchor to solve:

```
for acc in accs:
    if re.search(r"d[arp]$", acc):
        print("\t" + acc)
```

```
    45da
    de37dp
```

Double digest

This is a hard problem, and there are several ways to approach it. Let's simplify it by first figuring out what the fragment lengths would be if we digested the sequence with just a single restriction enzyme[1]. We'll open and read the file all in one go (there's no need to process it line-by-line as it's just a single sequence), then we'll use `re.finditer` to figure out the positions of all the cut sites.

The patterns themselves are relatively simple: N means any base, so the pattern for the AbcI site is `A[ATGC]TAAT`. The ambiguity code `R` means `A` or `G` and the code `W` means `A` or `T`, so the pattern for AbcII is `GC[AG][AT]TG`. Here's the code to calculate the start positions of the matches for AbcI:

[1] For the purposes of this exercise, we are of course ignoring all the interesting chemical kinetics of restriction enzymes and assuming that all enzymes cut with complete specificity and efficiency.

```
import re
dna = open("dna.txt").read().rstrip("\n")
print("AbcI cuts at:")
for match in re.finditer(r"A[ATGC]TAAT", dna):
    print(match.start())
```

The output from this looks good:

```
AbcI cuts at:
1140
1625
```

but it's not quite right – it's telling us the positions of the start of each match, but the enzyme actually cuts 3 base pairs upstream of the start. To get the position of the cut site, we need to add three to the start of each match:

```
import re
dna = open("dna.txt").read().rstrip("\n")
print("AbcI cuts at:")
for match in re.finditer(r"A[ATGC]TAAT", dna):
    print(match.start() + 3)
```

```
AbcI cuts at:
1143
1628
```

Now we've got the cut positions, how are we going to work out the fragment sizes? One way is to go through each cut site in order and measure the distance between it and the previous one – that will give us the length of a single fragment. To make this work we'll have to add "imaginary" cut sites at the very start and end of the sequence:

```
1 import re
2 dna = open("dna.txt").read().rstrip("\n")
3 all_cuts = [0]
4 for match in re.finditer(r"A[ATGC]TAAT", dna):
5     all_cuts.append(match.start() + 3)
6 all_cuts.append(len(dna))
7 print(all_cuts)
```

Let's take a moment to examine what's going on in this program. We start by creating a new list variable called `all_cuts` to hold the cut positions (line 3). At this point, the `all_cuts` variable only has one element: zero, the position of the start of the sequence. Next, for each match to the pattern (line 4), we take the start position, add three to it to get the cut position, and append that number to the `all_cuts` list (line 5). Finally, we append the position of the last character in the DNA string to the `all_cuts` list (line 6). When we print the `all_cuts` list, we can see that it contains the position of the start and end of the string, and the internal positions of the cut sites:

```
[0, 1143, 1628, 2012]
```

Now we can write a second loop to go through the all_cuts list and, for each cut position, work out the size of the fragment that will be created by figuring out the distance to the previous cut site (i.e. the previous element in the list). To make this work, however, we can't just use a normal loop – we have to start at the second element of the list (because the first element has no previous element) and we have to work with the index of each element, rather than the element itself. We'll use the range function to generate the list of indexes that we want to process – we need to go from index 1 (i.e. the second element of the list) to the last index (which is the length of the list):

```
1 for i in range(1,len(all_cuts)):
2     this_cut_position = all_cuts[i]
3     previous_cut_position = all_cuts[i-1]
4     fragment_size = this_cut_position - previous_cut_position
5     print("one fragment size is "  + str(fragment_size))
```

The loop variable `i` is used to store each value that is generated by the range function (line 1). For each value of `i` we get the cut position at that index (line 2) and the cut position at the previous index (line 3) and then figure out the distance between them (line 4). The output shows how, for two cuts, we get three fragments:

```
one fragment size is 1143
one fragment size is 485
one fragment size is 384
```

Now for the final part of the solution: how do we do the same thing for two different enzymes? We can add in the second enzyme pattern with the appropriate cut site offset and append the cut positions to the `all_cuts` variable:

```
import re
dna = open("dna.txt").read().rstrip("\n")
all_cuts = [0]

# add cut positions for AbcI
for match in re.finditer(r"A[ATGC]TAAT", dna):
    all_cuts.append(match.start() + 3)

# add cut positions for AbcII
for match in re.finditer(r"GC[AG][AT]TG", dna):
    all_cuts.append(match.start() + 4)

# add the final position
all_cuts.append(len(dna))
print(all_cuts)
```

but look what happens when we print the elements of all_cuts:

```
[0, 1143, 1628, 488, 1577, 2012]
```

We get zero, then the two cut positions for the first enzyme in ascending order, then the two cut positions for the second enzyme in ascending order, then the position of the end of the sequence. The method for turning a list of cut positions into fragment sizes that we developed above isn't going to work with this list, because it relies on the list of positions being in ascending order. If we try it with the list of cut positions produced by the above code, we'll end up with obviously incorrect fragment sizes:

```
one fragment size is 1143
one fragment size is 485
one fragment size is -1140
one fragment size is 1089
one fragment size is 434
```

Happily, Python's built-in `sort` function can come to the rescue. All we need to do is sort the list of cut positions before processing it, and we get the right answers. Here's the complete, final code:

```
import re
dna = open("dna.txt").read().rstrip("\n")
print(str(len(dna)))
all_cuts = [0]

# add cut positions for AbcI
for match in re.finditer(r"A[ATGC]TAAT", dna):
    all_cuts.append(match.start() + 3)

# add cut positions for AbcII
for match in re.finditer(r"GC[AG][AT]TG", dna):
    all_cuts.append(match.start() + 4)

# add the final position
all_cuts.append(len(dna))
sorted_cuts = sorted(all_cuts)
print(sorted_cuts)


for i in range(1,len(sorted_cuts)):
    this_cut_position = sorted_cuts[i]
    previous_cut_position = sorted_cuts[i-1]
    fragment_size = this_cut_position - previous_cut_position
    print("one fragment size is  "  + str(fragment_size))
```

8: Dictionaries

Storing paired data

Suppose we want to count the number of As in a DNA sequence. Carrying out the calculation is quite straightforward – in fact it's one of the first things we did in chapter 2:

```
dna = "ATCGATCGATCGTACGCTGA"
a_count = dna.count("A")
```

How will our code change if we want to generate a complete list of base counts for the sequence? We'll add a new variable for each base:

```
dna = "ATCGATCGATCGTACGCTGA"
a_count = dna.count("A")
t_count = dna.count("T")
g_count = dna.count("G")
c_count = dna.count("C")
```

and now our code is starting to look rather repetitive. It's not too bad for the four individual bases, but what if we want to generate counts for the 16 dinucleotides:

```
dna = "ATCGATCGATCGTACGCTGA"
aa_count = dna.count("AA")
at_count = dna.count("AT")
ag_count = dna.count("AG")
...etc. etc.
```

or the 64 trinucleotides:

```
dna = "ATCGATCGATCGTACGCTGA"
aaa_count = dna.count("AAA")
aat_count = dna.count("AAT")
aag_count = dna.count("AAG")
...etc. etc.
```

For trinucleotides and longer, the situation is particularly bad. The DNA sequence is 20 bases long, so it only contains 18 overlapping trinucleotides in total. This means that we'll end up with 64 different variables, at least 46 of which will hold the value zero.

One possible way round this is to store the values in a list. If we use three nested loops, we can generate all possible trinucleotides, calculate the count for each one, and store all the counts in a list:

```
dna = "AATGATCGATCGTACGCTGA"
all_counts = []
for base1 in ['A', 'T', 'G', 'C']:
    for base2 in ['A', 'T', 'G', 'C']:
        for base3 in ['A', 'T', 'G', 'C']:
            trinucleotide = base1 + base2 + base3
            count = dna.count(trinucleotide)
            print("count is " + str(count) + " for " + trinucleotide)
            all_counts.append(count)
print(all_counts)
```

Although the code is above is quite compact, and doesn't require huge numbers of variables, the output shows two problems with this approach:

```
count is 0 for AAA
count is 1 for AAT
count is 0 for AAG
count is 0 for AAC
count is 0 for ATA
count is 0 for ATT
count is 1 for ATG
count is 2 for ATC
 … many lines removed …
[0, 1, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
2, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0]
```

Firstly, the data are still incredibly sparse – the vast majority of the counts are zero. Secondly, the counts themselves are now disconnected from the trinucleotides. If we want to look up the count for a single trinucleotide – for example, TGA – we first have to figure out that TGA was the 25th trinucleotide generated by our loops. Only then can we get the element at the correct index:

```
print("count for TGA is " + str(all_counts[24]))
```

We can try various tricks to get round this problem. What if we generated two lists – one of counts, and one of the trinucleotides themselves?

```
dna = "AATGATCGATCGTACGCTGA"
all_trinucleotides = []
all_counts = []
for base1 in ['A', 'T', 'G', 'C']:
    for base2 in ['A', 'T', 'G', 'C']:
        for base3 in ['A', 'T', 'G', 'C']:
            trinucleotide = base1 + base2 + base3
            count = dna.count(trinucleotide)
            all_trinucleotides.append(trinucleotide)
            all_counts.append(count)
print(all_counts)
print(all_trinucleotides)
```

Now we have two lists of the same length, with a one-to-one correspondence between the elements:

```
[0, 1, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
2, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0]

['AAA', 'AAT', 'AAG', 'AAC', 'ATA', 'ATT', 'ATG', 'ATC', 'AGA', 'AGT',
'AGG', 'AGC', 'ACA', 'ACT', 'ACG', 'ACC', 'TAA', 'TAT', 'TAG', 'TAC',
'TTA', 'TTT', 'TTG', 'TTC', 'TGA', 'TGT', 'TGG', 'TGC', 'TCA', 'TCT',
'TCG', 'TCC', 'GAA', 'GAT', 'GAG', 'GAC', 'GTA', 'GTT', 'GTG', 'GTC',
'GGA', 'GGT', 'GGG', 'GGC', 'GCA', 'GCT', 'GCG', 'GCC', 'CAA', 'CAT',
'CAG', 'CAC', 'CTA', 'CTT', 'CTG', 'CTC', 'CGA', 'CGT', 'CGG', 'CGC',
'CCA', 'CCT', 'CCG', 'CCC']
```

This allows us to look up the count for a given trinucleotide in a slightly more appealing way – we can look up the index of the trinucleotide in the `all_trinucleotides` list, then get the count at the same index in the `all_counts` list:

```
i = all_trinucleotides.index('TGA')
c = all_counts[i]
print('count for TGA is ' + str(c))
```

This is a little bit nicer, but still has major drawbacks. We're still storing all those zeros, and now we have two lists to keep track of. We need to be incredibly careful when manipulating either of the two lists to make sure that they stay perfectly synchronized – if we make any change to one list but not the other, then there will no longer be a one-to-one correspondence between elements and we'll get the wrong answer when we try to look up a count.

This approach is also slow[1]. To find the index of a given trinucleotide in the `all_trinucleotides` list, Python has to look at each element one at a time until it finds the one we're looking for. This means that as the size of the list grows[2], the time taken to look up the count for a given element will grow alongside it.

If we take a step back and think about the problem in more general terms, what we need is a way of storing pairs of data (in this case, trinucleotides and their counts) in a way that allows us to efficiently look up the count for any given trinucleotide. This problem of storing paired data is incredibly common in programming. We might want to store:

- protein sequence names and their sequences
- DNA restriction enzyme names and their motifs
- codons and their associated amino acid residues
- colleagues' names and their email addresses
- sample names and their co-ordinates
- words and their definitions

[1] As a rule, we've avoided talking about performance in this book, but we'll break the rule in this case.

[2] For instance, imagine carrying out the same exercise with the approximately one million unique 10-mers.

All these are examples of what we call *key-value pairs*. In each case we have pairs of *keys* and *values*:

Key	**Value**
trinucleotide	count
name	protein sequence
name	restriction enzyme motif
codon	amino acid residue
name	email address
sample	coordinates
word	definition

The last example in this table – words and their definitions – is an interesting one because we have a tool in the physical world for storing this type of data – a dictionary. Python's tool for solving this type of problem is also called a dictionary (usually abbreviated to *dict*) and in this chapter we'll see how to create and use them.

Creating a dictionary

The syntax for creating a dictionary is similar to that for creating a list, but we use curly brackets rather than square ones. Each pair of data, consisting of a key and a value, is called an *item*. When storing items in a dictionary, we separate them with commas. Within an individual item, we separate the key and the value with a colon. Here's a bit of code that creates a dictionary of restriction enzymes (using data from the previous chapter) with three items:

```
enzymes = { 'EcoRI':r'GAATTC', 'AvaII':r'GG(A|T)CC', 'BisI':'GC[ATGC]GC' }
```

In this case, the keys and values are both strings[1]. Splitting the dictionary definition over several lines makes it easier to read:

```
enzymes = {
    'EcoRI' : r'GAATTC',
    'AvaII' : r'GG(A|T)CC',
    'BisI'  : r'GC[ATGC]GC'
}
```

but doesn't affect the code at all. To retrieve a bit of data from the dictionary – i.e. to look up the motif for a particular enzyme – we write the name of the dictionary, followed by the key in square brackets:

```
print(enzymes['BisI'])
```

The code looks very similar to using a list, but instead of giving the index of the element we want, we're giving the *key* for the *value* that we want to retrieve.

Dictionaries are a very useful way to store data, but they come with some restrictions. The only types of data we are allowed to use as keys are strings and numbers[2], so we can't, for example, create a dictionary where the keys are file objects. Values can be whatever type of data we like. Also, keys **must** be unique – we can't store multiple values for the same key.

In real-life programs, it's relatively rare that we'll want to create a dictionary all in one go like in the example above. More often, we'll want to create an empty dictionary, then add key/value pairs to it (just as we often create an empty list and then add elements to it).

To create an empty dictionary we simply write a pair of curly brackets on their own, and to add elements, we use the square-brackets notation on the left-hand side of

[1] The values are actually raw strings, but that's not important.

[2] Not strictly true; we can use any immutable type, but that is beyond the scope of this book.

an assignment. Here's a bit of code that stores the restriction enzyme data one item at a time:

```
enzymes = {}
enzymes['EcoRI'] = r'GAATTC'
enzymes['AvaII] =  r'GG(A|T)CC'
enzymes['BisI'] =  r'GC[ATGC]GC'
```

We can delete a key from a dictionary using the `pop` method. `pop` actually returns the value and deletes the key at the same time:

```
enzymes = {
    'EcoRI' : r'GAATTC',
    'AvaII' : r'GG(A|T)CC',
    'BisI'  : r'GC[ATGC]GC'
}
# remove the EcoRI enzyme from the dict
enzymes.pop('EcoRI')
```

Let's take another look at the trinucleotide count example from the start of the chapter. Here's how we store the trinucleotides and their counts in a dictionary:

```
dna = "AATGATCGATCGTACGCTGA"
counts = {}
for base1 in ['A', 'T', 'G', 'C']:
    for base2 in ['A', 'T', 'G', 'C']:
        for base3 in ['A', 'T', 'G', 'C']:
            trinucleotide = base1 + base2 + base3
            count = dna.count(trinucleotide)
            counts[trinucleotide] = count

print(counts)
```

We can see from the output that the trinucleotides and their counts are stored together in one variable:

```
{'ACC': 0, 'ATG': 1, 'AAG': 0, 'AAA': 0, 'ATC': 2, 'AAC': 0, 'ATA': 0,
'AGG': 0, 'CCT': 0, 'CTC': 0, 'AGC': 0, 'ACA': 0, 'AGA': 0, 'CAT': 0,
'AAT': 1, 'ATT': 0, 'CTG': 1, 'CTA': 0, 'ACT': 0, 'CAC': 0, 'ACG': 1,
'CAA': 0, 'AGT': 0, 'CAG': 0, 'CCG': 0, 'CCC': 0, 'CTT': 0, 'TAT': 0,
'GGT': 0, 'TGT': 0, 'CGA': 1, 'CCA': 0, 'TCT': 0, 'GAT': 2, 'CGG': 0,
'TTT': 0, 'TGC': 0, 'GGG': 0, 'TAG': 0, 'GGA': 0, 'TAA': 0, 'GGC': 0,
'TAC': 1, 'TTC': 0, 'TCG': 2, 'TTA': 0, 'TTG': 0, 'TCC': 0, 'GAA': 0,
'TGG': 0, 'GCA': 0, 'GTA': 1, 'GCC': 0, 'GTC': 0, 'GCG': 0, 'GTG': 0,
'GAG': 0, 'GTT': 0, 'GCT': 1, 'TGA': 2, 'GAC': 0, 'CGT': 1, 'TCA': 0,
'CGC': 1}
```

We still have a lot of repetitive counts of zero, but looking up the count for a particular trinucleotide is now very straightforward:

```
print(counts['TGA'])
```

We no longer have to worry about either "memorizing" the order of the counts or maintaining two separate lists.

Let's now see if we can find a way of avoiding storing all those zero counts. We can add an `if` statement that ensures that we only store a count if it's greater than zero:

```
dna = "AATGATCGATCGTACGCTGA"
counts = {}
for base1 in ['A', 'T', 'G', 'C']:
    for base2 in ['A', 'T', 'G', 'C']:
        for base3 in ['A', 'T', 'G', 'C']:
            trinucleotide = base1 + base2 + base3
            count = dna.count(trinucleotide)
            if count > 0:
                counts[trinucleotide] = count

print(counts)
```

When we look at the output from the above code, we can see that the amount of data we're storing is much smaller – just the counts for the trinucleotides that are greater than zero:

```
{'ATG': 1, 'ACG': 1, 'ATC': 2, 'GTA': 1, 'CTG': 1, 'CGC': 1, 'GAT': 2,
'CGA': 1, 'AAT': 1, 'TGA': 2, 'GCT': 1, 'TAC': 1, 'TCG': 2, 'CGT': 1}
```

Now we have a new problem to deal with. Looking up the count for a given trinucleotide works fine when the count is positive:

```
print(counts['TGA'])
```

But when the count is zero, the trinucleotide doesn't appear as a key in the dictionary:

```
print(counts['AAA'])
```

so we will get a KeyError when we try to look it up:

```
KeyError: 'AAA'
```

There are two possible ways to fix this. We can check for the existence of a key in a dictionary (just like we can check for the existence of an element in a list), and only try to retrieve it once we know it exists:

```
if 'AAA' in counts:
   print(counts('AAA'))
```

Alternatively, we can use the dictionary's `get` method. `get` usually works just like using square brackets: the following two lines do exactly the same thing:

```
print(counts['TGA'])
print(counts.get('TGA'))
```

The thing that makes `get` really useful, however, is that it can take an optional second argument, which is the default value to be returned if the key isn't present in the dictionary. In this case, we know that if a given trinucleotide doesn't appear in the dictionary then its count is zero, so we can give zero as the default value and use `get` to print out the count for any trinucleotide:

```
print("count for TGA is " + str(counts.get('TGA', 0)))
print("count for AAA is " + str(counts.get('AAA', 0)))
print("count for GTA is " + str(counts.get('GTA', 0)))
print("count for TTT is " + str(counts.get('TTT', 0)))
```

As we can see from the output, we now don't have to worry about whether or not each trinucleotide appears in the dictionary – `get` takes care of everything and returns zero when appropriate:

```
count for TGA is 2
count for AAA is 0
count for GTA is 1
count for TTT is 0
```

Iterating over a dictionary

What if, instead of looking up a single item from a dictionary, we want to do something for all items? For example, imagine that we wanted to take our counts dictionary variable from the code above and print out all trinucleotides where the count was 2. One way to do it would be to use our three nested loops again to generate all possible trinucleotides, then look up the count for each one and decide whether or not to print it:

```
for base1 in ['A', 'T', 'G', 'C']:
    for base2 in ['A', 'T', 'G', 'C']:
        for base3 in ['A', 'T', 'G', 'C']:
            trinucleotide = base1 + base2 + base3
            if counts.get(trinucleotide, 0) == 2:
                print(trinucleotide)
```

As we can see from the output, this works perfectly well:

```
ATC
TGA
TCG
GAT
```

But it seems inefficient to go through the whole process of generating all possible trinucleotides again, when the information we want – the list of trinucleotides – is already in the dictionary. A better approach would be to read the list of keys directly from the dictionary, which is what the `keys` method does.

Iterating over keys

When used on a dictionary, the `keys` method returns a list of all the keys in the dictionary:

```
print(counts.keys())
```

Looking at the output[1] confirms that this is the list of trinucleotides we want to consider (remember that we're looking for trinucleotides with a count of two, so we don't need to consider ones that aren't in the dictionary as we already know that they have a count of zero):

```
['ATG', 'ACG', 'ATC', 'GTA', 'CTG', 'CGC', 'GAT', 'CGA', 'AAT', 'TGA',
'GCT', 'TAC', 'TCG', 'CGT']
```

Using `keys`, our code for printing out all the trinucleotides that appear twice in the DNA sequence becomes a lot more concise:

```
for trinucleotide in counts.keys():
    if counts.get(trinucleotide) == 2:
        print(trinucleotide)
```

This version prints exactly the same set of trinucleotides as the more verbose method:

```
ATC
GAT
TGA
TCG
```

Before we move on, take a moment to compare the output immediately above this paragraph with the output from the three-loop version from earlier in this section. You'll notice that while the s**et of trinucleotides** is the same, the **order in which they appear** is different. This illustrates an important point about dictionaries – they are *inherently unordered*. That means that when we use the `keys` method to

[1] If you're using Python 3 you might see slightly different output here, but all the code examples will work just the same

iterate over a dictionary, we can't rely on processing the items in the same order that we added them. This is in contrast to lists, which always maintain the same order when looping. If we want to control the order in which keys are printed we can use the `sorted` method to sort the list before processing it:

```
for trinucleotide in sorted(counts.keys()):
    if counts.get(trinucleotide) == 2:
        print(trinucleotide)
```

Iterating over items

In the example code above, the first thing we need to do inside the loop is to look up the value for the current key. This is a very common pattern when iterating over dictionaries – so common, in fact, that Python has a special shorthand for it. Instead of doing this:

```
for key in my_dict.keys():
    value = my_dict.get(key)
    # do something with key and value
```

We can use the `items` method to iterate over pairs of data, rather than just keys:

```
for key, value in my_dict.items():
    # do something with key and value
```

The `items` method does something slightly different from all the other methods we've seen so far in this book; rather than returning a **single value**, or a **list of values**, it returns a **list of pairs of values**. That's why we have to give two variable names at the start of the loop. Here's how we can use the `items` method to process our dictionary of trinucleotide counts just like before:

```
for trinucleotide, count in counts.items():
    if count == 2:
        print(trinucleotide)
```

This method is generally preferred for iterating over items in a dictionary, as it makes the intention of the code very clear.

Recap

We started this chapter by examining the problem of storing paired data in Python. After looking at a couple of unsatisfactory ways to do it using tools that we've already learned about, we introduced a new type of data structure – the dictionary – which offers a much nicer solution to the problem of storing paired data.

Later in the chapter, we saw that the real benefit of using dictionaries is the efficient lookup they provide. We saw how to create dictionaries and manipulate the items in them, and several different ways to look up values for known keys. We also saw how to iterate over all the items in dictionary.

In the process, we uncovered a few restrictions on what dictionaries are capable of – we're only allowed to use a couple of different data types for keys, they must be unique, and we can't rely on their order. Just as a physical dictionary allows us to rapidly look up the definition for a word but not the other way round, Python dictionaries allow us to rapidly look up the value associated with a key, but not the reverse.

Exercises

DNA translation

Write a program that will translate a DNA sequence into protein. Your program should use the standard genetic code which can be found at this URL[1].

[1] http://www.ncbi.nlm.nih.gov/Taxonomy/taxonomyhome.html/index.cgi?chapter=tgencodes#SG1

Solutions

DNA translation

The description of this exercise is very short, but it hides quite a bit of complexity! To translate a DNA sequence we need to carry out a number of different steps. First, we have to split up the sequence into codons. Then, we need to go through each codon and translate it into the corresponding amino acid residue. Finally, we need to create a protein sequence by adding all the amino acid residues together.

We'll start off by figuring out how to split a DNA sequence into codons. Because this exercise is quite tricky, we'll pick a very short test DNA sequence to work on – just three codons:

```
dna = "ATGTTCGGT"
```

How are we going to split up the DNA sequence into groups of three bases? It's tempting to try to use the `split` method, but remember that the `split` method only works if the things you want to split are separated by a delimiter. In our case, there's nothing separating the codons, so `split` will not help us.

Something that might be able to help us is substring notation. We know that this allows us to extract part of a string, so we can do something like this:

```
dna = "ATGTTCGGT"
codon1 = dna[0:3]
codon2 = dna[3:6]
codon3 = dna[6:9]
print(codon1, codon2, codon3)
```

As we can see from the output, this works:

```
('ATG', 'TTC', 'GGT')
```

but it's not a great solution, as we have to fill in the numbers manually. Since the numbers follow a very predictable pattern, it should be possible to generate them automatically. The start position for each substring is initially zero, then goes up by three for each successive codon. The stop position is just the start position plus three.

Recall that the job of the `range` function is to generate sequences of numbers. In order to generate the sequence of substring start positions, we need to use the three-argument version of `range`, where the first argument is the number to start at, the second argument is the number to finish at, and the third argument is the step size. For our DNA sequence above, the number to start at is zero, and the step size is three. The number to finish at it not six but seven, because ranges are exclusive at the finish. This bit of code shows how we can use the `range` function to generate the list of start positions:

```
for start in range(0,7,3):
   print(start)
```

```
0
3
6
```

To find the stop position for a given start position we just add three, so we can easily split our DNA into codons using a loop:

```
dna = "ATGTTCGGT"
for start in range(0,7,3):
    codon = dna[start:start+3]
    print("one codon is" + codon)
```

```
one codon is ATG
one codon is TTC
one codon is GGT
```

This works fine for our test DNA sequence, but if we give it a shorter sequence we will get incomplete and empty codons:

```
dna = "ATGTT"
for start in range(0,7,3):
    codon = dna[start:start+3]
    print(codon)
```

```
one codon is ATG
one codon is TT
one codon is
```

and if we give it a longer sequence, we will miss out the fourth and subsequent codons:

```
dna = "ATGTTCGGTGAAGCGGGCTAGAT"
for start in range(0,7,3):
    codon = dna[start:start+3]
    print("one codon is " + codon)
```

```
one codon is ATG
one codon is TTC
one codon is GGT
```

Clearly we need to modify the second argument to `range` – the position to finish the sequence of numbers – in order to take into account the length of the DNA sequence. At this point, we have to confront the problem of what to do if we're given a DNA sequence whose length is not an exact multiple of three. Clearly, we cannot translate an incomplete codon, so we want the start position of the final codon to equal to the length of the DNA sequence minus two. This guarantees that there will always be two more characters following the position of the final codon start – i.e. enough for a complete codon.

Here's the modified code:

```
dna = "ATGTTCGGT"

# calculate the start position for the final codon
last_codon_start = len(dna) - 2

# process the dna sequence in three base chunks
for start in range(0,last_codon_start,3):
    codon = dna[start:start+3]
    print("one codon is " + codon)
```

Now that we know how to split a DNA sequence up into codons, let's turn our attention to the problem of translating those codons. If we pull up the URL from the exercise description in a web browser, we can see the standard codon translation table in various formats. Storing this translation table seems like a perfect job for a dictionary: we have codons (keys) and amino acid residues (values) and we want to be able to look up the amino acid for a given codon.

Here's a bit of code – it's actually a single statement, spread out over multiple lines – which creates a dictionary to hold the translation table:

```
gencode = {
'ATA':'I', 'ATC':'I', 'ATT':'I', 'ATG':'M',
'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACT':'T',
'AAC':'N', 'AAT':'N', 'AAA':'K', 'AAG':'K',
'AGC':'S', 'AGT':'S', 'AGA':'R', 'AGG':'R',
'CTA':'L', 'CTC':'L', 'CTG':'L', 'CTT':'L',
'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCT':'P',
'CAC':'H', 'CAT':'H', 'CAA':'Q', 'CAG':'Q',
'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGT':'R',
'GTA':'V', 'GTC':'V', 'GTG':'V', 'GTT':'V',
'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCT':'A',
'GAC':'D', 'GAT':'D', 'GAA':'E', 'GAG':'E',
'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGT':'G',
'TCA':'S', 'TCC':'S', 'TCG':'S', 'TCT':'S',
'TTC':'F', 'TTT':'F', 'TTA':'L', 'TTG':'L',
'TAC':'Y', 'TAT':'Y', 'TAA':'_', 'TAG':'_',
'TGC':'C', 'TGT':'C', 'TGA':'_', 'TGG':'W'}
```

We can look up the amino acid for a given codon using either of the two methods that we learned about:

```
print(gencode['CAT'])
print(gencode.get('GTC'))
```

```
H
V
```

If we look up the amino acid for each codon inside the loop of our original code, we can print both the codon and the amino acid translation[1]:

```
dna = "ATGTTCGGT"
last_codon_start = len(dna) - 2
for start in range(0,last_codon_start,3):
    codon = dna[start:start+3]
    aa = gencode.get(codon)
    print("one codon is " + codon)
    print("the amino acid is " + aa)
```

```
one codon is ATG
the amino acid is M
one codon is TTC
the amino acid is F
one codon is GGT
the amino acid is G
```

This is starting to look promising. The final step is to actually do something with the amino acid residues rather than just printing them. A nice idea is to take our cue from the way that a ribosome behaves and add each new amino acid residue onto the end of a protein to create a gradually-growing string:

[1] From now on, we won't include the statement which creates the dictionary in our code samples as it takes up too much room, so if you want to try running these yourself you'll need to add it back at the top.

```
1 dna = "ATGTTCGGT"
2 last_codon_start = len(dna) - 2
3 protein = ""
4 for start in range(0,last_codon_start,3):
5     codon = dna[start:start+3]
6     aa = gencode.get(codon)
7     protein = protein + aa
8 print("protein sequence is " + protein)
```

In the above code, we create a new variable to hold the protein sequence immediately before we start the loop (line 3), then add a single character onto the end of that variable each time round the loop (line 7). By the time we exit the loop, we have built up the complete protein sequence and we can print it out (line 8):

```
protein sequence is MFG
```

This looks like a very useful bit of code, so let's turn it into a function. Our function will take one argument – the DNA sequence as a string – and will return a string containing the protein sequence[1]:

```
def translate_dna(dna):
    last_codon_start = len(dna) - 2
    protein = ""
    for start in range(0,last_codon_start,3):
        codon = dna[start:start+3]
        aa = gencode.get(codon)
        protein = protein + aa
    return protein
```

We can now test our function by printing out the protein translation for a few more test sequences:

[1] You'll notice that this function relies on the `gencode` variable which is defined outside the function – something that I told you not to do in chapter 5. This is an exception to the rule: defining the gencode variable inside the function means that it would have to be created anew each time we wanted to translate a DNA sequence.

```
print(translate_dna("ATGTTCGGT"))
print(translate_dna("ATCGATCGATCGTTGCTTATCGATCAG"))
print(translate_dna("actgatcgtagctagctgacgtatcgtat"))
print(translate_dna("ACGATCGATCGTNACGTACGATCGTACTCG"))
```

The output from this code shows that we run into a problem with the third sequence:

```
MFG
IDRSLLIDQ
Traceback (most recent call last):
  File "dna_translation.py", line 30, in <module>
    print(translate_dna("actgatcgtagctagctgacgtatcgtat"))
  File "dna_translation.py", line 25, in translate_dna
    protein = protein + aa
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

The problem occurs when we try to look up the amino acid for the first codon of the third sequence – "act". Because the third sequence is in lower case but the translation table dictionary is in upper case, the key isn't found, the `get` method returns `None`, and we get an error. Fixing it is straightforward – we just need to convert the codon to upper case before looking up the amino acid:

```
def translate_dna(dna):
    last_codon_start = len(dna) - 2
    protein = ""
    for start in range(0,last_codon_start,3):
        codon = dna[start:start+3]
        aa = gencode.get(codon.upper())
        protein = protein + aa
    return protein
```

Now the output shows that the first three sequences are fine, but that our function has a problem translating the fourth sequence:

```
MFG
IDRSLLIDQ
TDRSLLTYR
Traceback (most recent call last):
  File "dna_translation.py", line 31, in <module>
    print(translate_dna("ACGATCGATCGTNACGTACGATCGTACTCG"))
  File "dna_translation.py", line 25, in translate_dna
    protein = protein + aa
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

Glancing at the input sequences, it's not clear what the problem is. Let's try printing the codons as they're translated in order to identify the one that's causing the error:

```
def translate_dna(dna):
    last_codon_start = len(dna) - 2
    protein = ""
    for start in range(0,last_codon_start,3):
        codon = dna[start:start+3]
        print("about to translate codon: " + codon)
        aa = gencode.get(codon.upper())
        protein = protein + aa
    return protein

print(translate_dna("ACGATCGATCGTNACGTACGATCGTACTCG"))
```

The output shows where the problem lies:

```
about to translate codon: ACG
about to translate codon: ATC
about to translate codon: GAT
about to translate codon: CGT
about to translate codon: NAC
Traceback (most recent call last):
  File "dna_translation.py", line 32, in <module>
    print(translate_dna("ACGATCGATCGTNACGTACGATCGTACTCG"))
  File "dna_translation.py", line 26, in translate_dna
    protein = protein + aa
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

There is an unknown base in the middle of the DNA sequence, which causes our function to try to look up the amino acid for the codon `NAC`, which causes an error because that codon isn't in the dictionary. How should we fix this? We could add an `if` statement to the function which only translates the DNA sequence if it doesn't contain any unambiguous bases, but that seems a little too conservative – there are plenty of situations in which we might want to generate a protein sequence for a DNA sequence that has unknown bases. We could add an `if` statement inside the loop which only translates a given codon if it doesn't contain any unambiguous bases, but that would lead to protein translations of an incorrect length – we know that the codon NAC will translate to an amino acid, we just don't know which one it will be.

The most sensible solution seems to be to translate any codon with an unknown base into the symbol for an unknown amino acid residue, which is `X`. The optional second argument to the `get` function makes it very easy to do just that:

```
def translate_dna(dna):
    last_codon_start = len(dna) - 2
    protein = ""
    for start in range(0,last_codon_start,3):
        codon = dna[start:start+3]
        aa = gencode.get(codon.upper(), 'X')
        protein = protein + aa
    return protein
```

and now we can translate all four of our test sequences correctly:

```
print(translate_dna("ATGTTCGGT"))
print(translate_dna("ATCGATCGATCGTTGCTTATCGATCAG"))
print(translate_dna("actgatcgtagcttgcttacgtatcgtat"))
print(translate_dna("ACGATCGATCGTNACGTACGATCGTACTCG"))
```

```
MFG
IDRSLLIDQ
TDRSLLTYR
TIDRXVRSYS
```

At this point, it's a good idea to turn these test sequences into `assert` statements – that way, we can easily re-test the function if we make some changes to it in the future:

```
assert(translate_dna("ATGTTCGGT")) == "MFG"
assert(translate_dna("ATCGATCGATCGTTGCTTATCGATCAG")) == "IDRSLLIDQ"
assert(translate_dna("actgatcgtagcttgcttacgtatcgtat")) == "TDRSLLTYR"
assert(translate_dna("ACGATCGATCGTNACGTACGATCGTACTCG")) == "TIDRXVRSYS"
```

9: Files, programs, and user input

File contents and manipulation

Reading from and writing to files was one of the first things we looked at in this book, back in chapter 3. For some programs, however, we're not just concerned with the contents of files, but with files and folders themselves. This is especially likely to be the case for programs that have to operate as part of a work flow involving other tools and software. For example, we may need to copy, move, rename and delete files, or we may need to process all files in a certain folder.

Although it seems like a simple task (after all, the file manager tools that come with your operating system can carry most of them out), file manipulation in a language like Python is actually quite tricky. That's because the code that we write has to function identically on different operating systems – including Windows, Linux and Mac machines – which may handle files quite differently. A discussion of the differences between operating systems is way beyond the scope of this book, but to give one example, UNIX-based systems like Linux and OSX have the concept of file permissions which is lacking in Windows.

Thankfully, Python includes a couple of modules[1] that take care of these differences for us and provide us with a set of useful functions for manipulating files. The modules' names are `os` (short for Operating System) and `shutil` (short for SHell UTILities). In the next section we'll see how they can be used to carry out various common (but important) tasks.

A note on the code examples

Since the code examples in this chapter unavoidably involve interaction with the operating system, some of the details will be operating-system specific. In

[1] Take a look back at chapter 7 for a reminder of how modules work.

particular, many of the file manipulation functions take *paths* as arguments, which differ considerably between operating systems. A *path* is the short bit of text that tells you the location of a file in the file system. On Linux and OSX machines, the path to a file or folder typically looks like this:

```
/path/to/my/file.txt
```

whereas on Windows machines, they look like this:

```
c:\path\to\my\file.txt
```

Moreover, the success of the code examples for many functions relies on the files and folders actually being present on the computer on which the examples are run. The code examples in this chapter will use Linux-style paths, and will refer to folders and files on my computer, so if you want to try running them, you'll probably need to change the paths to refer to files on your own computer.

Basic file manipulation

To rename an existing file, we simply import the `os` module, then use the `os.rename` function. The `os.rename` function takes two arguments, both strings. The first is the current name of the file, the second is the new name:

```
import os
os.rename("old.txt", "new.txt")
```

The above code assumes that the file *old.txt* is in the folder where we are running our Python program. If it's elsewhere in the filesystem, then we have to give the complete path:

```
os.rename("/home/martin/biology/old.txt", "/home/martin/biolgy/new.txt")
```

If we specify a different folder, but the same file name, in the second argument, then the function will move the file from one folder to another:

```
os.rename("/home/martin/biology/old.txt", "/home/martin/python/old.txt")
```

Of course, we can move and rename a file in one step if you like:

```
os.rename("/home/martin/biology/old.txt", "/home/martin/python/new.txt")
```

`os.rename` works on folders as well as files:

```
os.rename("/home/martin/old_folder", "/home/martin/new_folder")
```

If we try to move a file to a folder that doesn't exist we'll get an error. We need to create the new folder first with the `os.mkdir` function:

```
os.mkdir("/home/martin/python")
```

If we need to create a bunch of directories all in one go, we can use the `os.mkdirs` function (note the s on the end of the name):

```
os.mkdir("/a/long/path/with/lots/of/folders")
```

To copy a file or folder we use the `shutil` module. We can copy a single file with `shutil.copy`:

```
shutil.copy("/home/martin/original.txt", "/home/martin/copy.txt")
```

or a folder with `shutil.copytree`:

```
shutil.copytree("/home/martin/original_folder",
"/home/martin/copy_folder")
```

To test whether a file or folder exists, use `os.path.exists`:

```
if os.path.exists("/home/martin/email.txt"):
    print("You have mail!")
```

Deleting files and folders

There are different functions for deleting files, empty folders, and non-empty folders. To delete a single file, use `os.remove`:

```
os.remove("/home/martin/unwanted_file.txt")
```

To delete an empty folder, use `os.rmdir`:

```
os.rmdir("/home/martin/emtpy")
```

To delete a folder and all the files in it, use `shutil.rmtree`

```
shutil.rmtree("home/martin/full")
```

Listing folder contents

The `os.listdir` function returns a list of files and folders. It takes a single argument which is a string containing the path of the folder whose contents you want to search. To get a list of the contents of the current working directory, use the string "." for the path:

```
for file_name in os.listdir("."):
   print("one file name is " + file_name)
```

To list the contents of a different folder, we just give the path as an argument:

```
for file_name in os.listdir("/home/martin"):
   print("one file name is " + file_name)
```

Running external programs

Another feature of Python that involves interaction with the operating system is the ability to run external programs. Just like file and folder manipulation, the ability to run other programs is very useful when using Python as part of a work flow. It allows us to use existing tools that would be very time-consuming to recreate in Python, or that would run very slowly.

Running external programs from within your Python code can be a tricky business, and this feature wouldn't normally be covered in an introductory programming course. However, it's so useful for biology (and science in general) that we're going to cover it here, albeit in a simplified form.

As with the above section on file operations, the exact details of how external programs are run will vary with your operating system and the way your computer is set up. On UNIX-based systems, the program that you want to run might already be in your path, in which case you can simply use the name of the executable as the string to be executed. For the example code below, I'll give the full path to executables on my computer, which look something like this:

```
/home/martin/software/myprogram
```

If you're on Windows, your paths will probably look like this:

```
c:\windows\Program files\myprogram\myprogram.exe
```

And on OSX, they will look like this:

```
/Applications/myprogram
```

As before, if you want to try running any of these examples, make sure that you change the paths to point to real executables on your computer.

Running a program

The functions for running external program reside in the `subprocess` module. The reasoning behind the name is slightly convoluted: when talking about operating systems, a running program is called a *process*, and a program that is started by another program is called a *subprocess*.

To run an external program, use the `subprocess.call` function. This function takes a single string argument containing the path to the executable you want to run:

```
import subprocess
subprocess.call("/bin/date")
```

Any output that is produced by the external program is printed straight to the screen – in this case, the output from the Linux `date` program:

```
Fri Jul 26 15:15:26 BST 2013
```

If we want to supply command-line options to the external program then we just include them in the string, and set the optional `shell` argument to `True`. Here we call the Linux `date` program with the options which cause it to just print the month:

```
subprocess.call("/bin/date +%B", shell=True)
```

```
July
```

Saving program output

Often, we want to run some external program and then store the output in a variable so that we can do something useful with it. For this, we use `subprocess.check_output`, which takes exactly the same arguments as `subprocess.call`:

```
current_month = subprocess.check_output("/bin/date +%B", shell=True)
```

Just like when reading file contents, the output from an external program can run over multiples lines that end with new line characters, so you probably need to use `rstrip` to remove them before carrying out any processing.

User input makes our programs more flexible

The exercises and examples that we've seen so far in this book have used two different ways of getting date into a program. For small bits of data, like short DNA sequences, restriction enzyme motifs, and gene accession names, we've simply stored the data directly in a variable like this:

```
dna = "ATCGATCGTGACTAGCTACG"
```

When data is mixed in with the code in this manner, it is said to be *hard-coded*.

For larger pieces of data, like longer DNA sequences and spreadsheet-like data, we've typically read the information from an external text file. For many purposes, this is a better solution than hard-coding the data, as it allows the separation of data and code, making our programs easier to read. However, in all the examples we've seen so far, the *names* of the files from which the data are read are still hard-coded.

Both of these approaches to getting data in to our program have the same shortcomings – if we want to change the input data, we have to open up the code and edit it. In the case of hard-coded variables, we have to edit the statement where the variables are created. In the case of files, we have two choices – we can either edit the contents of the file, or edit the hard-coded file name.

Real-life useful programs don't generally work that way. Instead, they generally allow us to specify input files and options at the time when we run the program, rather than when we're writing it. This allows programs to be much more flexible and easier to use, especially for a person who didn't write the code in the first place.

In the next couple of sections we're going to see a couple of tools for getting user input, but more importantly we're going to talk about the transition from writing a program that's only useful to you, to writing one that can be used by other people. This involves starting to think about the experience of using a program from the perspective of a user.

There are many reasons why you might need your programs to be usable by somebody who's not familiar with the code. If you write a program that solves a problem for you, chances are that it could solve a problem for your colleagues and collaborators as well. If you write a program that forms a significant part of a piece of work which you later want to publish, you many have to make sure that whoever is peer-reviewing your paper can get your program working as well. Of course, making your program easier to use for other people means that it will also be easier to use for you, a few months after you have written it when you have completely forgotten how the code works!

Interactive user input

To get interactive input from the user in our programs, we can use the `input` function. `input` takes a single string argument, which is the prompt to be displayed to the user, and returns the value typed in as a string:

```
accession = input("Enter the accession name")
# do something with the accession variable
```

The `input` function behaves a little differently to other functions and methods we've seen, because it has to wait for something to happen before it can return a value – the user has to type in a string and press enter. The user input will be returned as a string (so if we need to use is as something else – e.g. a number – we'll have to do the conversion manually) and will end with a new line (so we might want to use `rstrip` to remove it).

Capturing user input in this way requires us to think quite carefully about how our program behaves. Programs that we write to carry out analysis of large datasets will often take a considerable amount of time to run, so it's important that we minimize the chances of the user having to re-run them. When using the `input` function, there are two situations in particular that we want to avoid.

One is the situation where we have a long-running program that requires some user input, but doesn't make this fact clear to the user. What can happen in this scenario is that the user starts the program running and then switches their attention to something else, assuming that the program will continue to make progress in the background. If the user doesn't notice (or is not at their computer) when the program reaches the point where it requires input and halts, the program may be stuck waiting for input for a long time.

The other scenario to avoid is that where a program runs for some time before asking the user for input, then fails to work due to an incorrect input or typo, requiring the user to re-start the program from scratch.

A good way to avoid both of these problems is to design our programs such that they collect all necessary user input at the start, before any long-running tasks are carried out. We can also reduce the chances of incorrect input on the part of the user by offering clear instructions and documentation.

An important part of user input is *input validation* – checking that the input supplied by the user makes sense. For example, you might require that a particular input is a number between some minimum and maximum values, or that it's a DNA sequence without ambiguous bases, or that it's the name of a file that must exist. A good strategy for input validation is to check the input as soon as it's received, and give the user a second chance to enter their input if it's found to be invalid.

One big drawback of getting user input interactively is that it makes it harder to run a program unsupervised as part of a work flow. For most biological analyses, specifying program options when it's run using command line arguments is a better approach.

Command line arguments

If you're used to using existing programs that have a command-line user interface (as opposed to a graphical one) then you're probably familiar with command line arguments[1]. These are the strings that you type on the command line after the name of a program you want to run:

```
myprogram one two three
```

In the above code, one two and three are the command line options. To use command line arguments in our Python scripts, we import the `sys` module. We can then access the command line arguments by using the special list `sys.argv.` Running the following code:

[1] Not to be confused with the arguments that we give to functions, although they do a similar job.

```
import sys
print(sys.argv)
```

with the command line:

```
 python myprogram.py one two three
```

shows how the elements of `sys.argv` are made up of the arguments given on the command line:

```
['myprogram.py', 'one', 'two', 'three']
```

Note that the first element of `sys.argv` is always the name of the program itself, so the first command line argument is at index one, the second at index two, etc.

Just like with `input`, options and filenames given on the command line are stored as strings, so if, for example, we want to use a command line argument as a number, we'll have to convert it with `int`.

Command line arguments are a good way of getting input for your Python programs for a number of reasons. All the data your program needs will be present at the start of your program, so you can do any necessary input validation (like checking that files are present) before starting any processing. Also, your program will be able to be run as part of a shell script, and the options will appear in the user's shell history.

Recap

We started this chapter by examining two features of Python that allow your programs to interact with the operating system – file manipulation and external processes. We learned which functions to use for common file system operations, and which modules they belong to. We also ssaw two ways to call external programs from within your Python program.

When using these techniques to solve real life problems, or when working on the exercises, remember that you may encounter errors that are nothing to do with your program. For instance, when trying to manipulate files you may get an error if a specified file doesn't exist or you don't have the necessary permissions to rename it. Similarly, if you get unexpected output when running an external program the problem may lie with the external program or with the way that you're calling it, rather than with your Python program. This is in contrast to the rest of the exercises in this book, which are mostly self-contained. If you run into difficulties when using the tools in this chapter, check the external factors as well as checking your program code.

In the last portion of the chapter, we saw two different ways to get user input when your program runs. Using command line arguments is generally better for the type of programming that forms part of scientific research.

Exercises

In the *chapter_9* folder in the exercises download there is a collection of files with the extension *.dna* which contain DNA sequences of varying length, one per line. Use this set of files for both exercises.

Binning DNA sequences

Write a program which creates nine new folders – one for sequences between 100 and 199 bases long, one for sequences between 200 and 299 bases long, etc. Write out each DNA sequence in the input files to a separate file in the appropriate folder.

Kmer counting

Write a program that will calculate the number of all kmers of a given length across all DNA sequences in the input files and display just the ones that occur more than a given number of times. You program should take two command line arguments – the kmer length, and the cutoff number.

Solutions

Binning DNA sequences

The first job is to figure out how to read all the DNA sequences. We can get a list of all the files in the folder by using `os.listdir`, but we'll have to be careful to only read DNA sequences from files that have the right file name extension. Here's a bit of code to start off with:

```
import os

for file_name in os.listdir("."):
    if file_name.endswith(".dna"):
        print("reading sequences from " + file_name)
```

We can check the output to make sure that we're only going to process the correct files:

```
reading sequences from xag.dna
reading sequences from xaj.dna
reading sequences from xaa.dna
reading sequences from xab.dna
reading sequences from xai.dna
reading sequences from xae.dna
reading sequences from xah.dna
reading sequences from xaf.dna
reading sequences from xac.dna
reading sequences from xad.dna
```

The next step is to read the DNA sequences from each file. For each file that passes the name test, we'll open it, then process it one line at a time and calculate the length of the DNA sequence:

```
# look at each file
for file_name in os.listdir("."):
    if file_name.endswith(".dna"):
        print("reading sequences from " + file_name)
        dna_file = open(file_name)

        # look at each line
        for line in dna_file:
            dna = line.rstrip("\n")
            length = len(dna)
            print("found a dna sequence with length " + str(length))
```

Notice how we've used rstrip to remove the new line character – we don't want to include it in the count of the sequence length, since it's not a base. With ten files, and ten DNA sequences per file, this program generates over a hundred lines of output – here's the first few:

```
reading sequences from xag.dna
found a dna sequence with length 432
found a dna sequence with length 818
found a dna sequence with length 604
found a dna sequence with length 879
found a dna sequence with length 619
found a dna sequence with length 500
found a dna sequence with length 119
found a dna sequence with length 341
found a dna sequence with length 303
found a dna sequence with length 469
reading sequences from xaj.dna
found a dna sequence with length 121
found a dna sequence with length 442
found a dna sequence with length 520
```

This looks good – we're getting a range of different sizes. Next we have to figure out which bin each of the sequences should go in. Because the limits of the bins follow a regular pattern, we can use the `range` function to generate them. We can generate a list of the lower limits for each bin by taking a range of numbers from

100 to 1000 with a step size of 100, then adding 99 to get the upper limit of the bin. We'll go through this process for each sequence, checking if it belongs in each bin in turn:

```
# go through each file in the folder
for file_name in os.listdir("."):

  # check if it ends with .dna
  if file_name.endswith(".dna"):
    print("reading sequences from " + file_name)

    # open the file and process each line
    dna_file = open(file_name)
    for line in dna_file:

      # calculate the sequence length
      dna = line.rstrip("\n")
      length = len(dna)
      print("sequence length is " + str(length))

      # go through each bin and check if the sequence belongs in it
      for bin_lower in range(100,1000,100):
        bin_upper = bin_lower + 99
        if length >= bin_lower and length < bin_upper:
          print("bin is " + str(bin_lower) + " to " + str(bin_upper))
```

There are quite a few levels of indentation in the above code, so you might have to read it through a few times. We have

- the loop for each file name
- the if statement that checks the file name
- the loop for each sequence in a file
- the loop for each bin
- the if statement that checks if the sequence belongs in the bin

The first few lines of the output show that this approach works:

```
reading sequences from xag.dna
sequence length is 432
bin is 400 to 499
sequence length is 818
bin is 800 to 899
sequence length is 604
bin is 600 to 699
sequence length is 879
bin is 800 to 899
sequence length is 619
bin is 600 to 699
sequence length is 500
bin is 500 to 599
sequence length is 119
```

The final step is to create the new folders, and write each DNA sequence to the appropriate one. We can re-use our `range` idea to generate the folder names and create them. The name of the folder for a given bin is the lower limit, followed by an underscore, followed by the upper limit:

```
for bin_lower in range(100,1000,100):
    bin_upper = bin_lower + 99
    bin_folder_name = str(bin_lower) + "_" + str(bin_upper)
    os.mkdir(bin_folder_name)
```

When we want to write out DNA sequence to a file in a particular folder, we can use the same naming scheme to work out the name of the folder. Of course, we also have to figure out what to call the individual files of DNA sequences. The exercise description didn't specify any kind of naming scheme, so we'll keep things simple and store the first DNA sequence in a file called *1.dna*, the second in a file called *2.dna*, etc. We'll need to create an extra variable to hold the number of DNA sequences we've seen, and to increment it after writing each DNA sequence. Here's the whole script – it's by far the largest program that we've written so far:

```
import os

# create a new folder for each bin
 for bin_lower in range(100,1000,100):
    bin_upper = bin_lower + 99
    bin_folder_name = str(bin_lower) + "_" + str(bin_upper)
    os.mkdir(bin_folder_name)

# create a variable to hold the sequence number
seq_number = 1

# process all files that end in .dna
for file_name in os.listdir("."):
  if file_name.endswith(".dna"):
    print("reading sequences from " + file_name)
    dna_file = open(file_name)

    # for each line, calculate the sequence length
    for line in dna_file:
      dna = line.rstrip("\n")
      length = len(dna)
      print("sequence length is " + str(length))

      # figure out which bin the sequence belongs in
      for bin_lower in range(100,1000,100):
        bin_upper = bin_lower + 99
        if length >= bin_lower and length < bin_upper:

          # once we know the correct bin, write out the sequence
          print("bin is " + str(bin_lower) + " to " + str(bin_upper))
          bin_folder_name = str(bin_lower) + "_" + str(bin_upper)
          output_path = bin_folder_name + '/' + str(seq_number) + '.dna'
          output = open(output_path, "w")
          output.write(dna)
          output.close()

          # increment the sequence number
          seq_number = seq_number+1
```

Kmer counting

To come up with a plan of attack for this exercise, we must first think about the order in which we process the data. Can we simply read a single DNA sequence, count the k-mers, and print the counts like we did for the trinucleotide example in chapter 8? No, because we only want to print the k-mers which occur more than a given number of times across **all** sequences. In other words, we don't know which k-mers we want to print the counts for until we have finished processing all sequences.

So, we will have to tackle this problem in two stages. First, we will go through each sequence one-by-one and gradually build up a list of k-mer counts. Second, we will go through the list of counts and print only the ones whose count is above the cutoff.

How will we generate the k-mer counts? A good first step would be to figure out how to split a DNA sequence into overlapping k-mers of any given length. We can use a similar approach to the one taken in the DNA translation exercise in chapter 8: use the range function to generate a list of the start positions of each k-mer, then use substring notation to extract the k-mer from the sequence. Here's a bit of code that prints all k-mers of a given size. We'll use a short test DNA sequence for now:

```
test_dna = "ACTGTAGCTGTACGTAGC"
print(test_dna)
kmer_size = 4
for start in range(0,len(test_dna)-(kmer_size-1),1):
    kmer = test_dna[start:start+kmer_size]
    print(kmer)
```

The tricky bit is figuring out the arguments to the range function. We know that we want to start at zero and increase by one each time. The finish position is the length of the sequence, minus the k-mer size (to make sure there is one k-mer's worth of bases after it) minus one (to allow for the fact that the finish position is exclusive). The range function generates the start positions for each k-mer, and to get the end positions we just add the k-mer size. We can examine the output from this code and check that it agrees with out intuition:

```
ACTGTAGCTGTACGTAGC
ACTG
CTGT
TGTA
GTAG
TAGC
AGCT
GCTG
CTGT
TGTA
GTAC
TACG
ACGT
CGTA
GTAG
TAGC
```

To make it easier to test this bit of code, we'll turn it into a function. The function will take two arguments. The first argument will be the DNA sequence as a string, and the second argument will be the k-mer size as a number. Instead of printing the list of k-mers, it will return a list of them. Here's the code for the function and three statements to test it:

```
def split_dna(dna, kmer_size):
    kmers = []
    for start in range(0,len(dna)-(kmer_size-1),1):
        kmer = dna[start:start+kmer_size]
        kmers.append(kmer)
    return kmers


print(split_dna("AATGCTGCAT", 4))
print(split_dna("AATGCTGCAT", 5))
print(split_dna("AATGCTGCAT", 6))
```

As we can see from the output, running the function multiple times with the same DNA sequence but different k-mer lengths gives different results, as expected:

```
['AATG', 'ATGC', 'TGCT', 'GCTG', 'CTGC', 'TGCA', 'GCAT']
['AATGC', 'ATGCT', 'TGCTG', 'GCTGC', 'CTGCA', 'TGCAT']
['AATGCT', 'ATGCTG', 'TGCTGC', 'GCTGCA', 'CTGCAT']
```

Now we can put this function together with the code we developed for looping through files from the previous exercise. To count up the k-mers, we will create an empty dictionary at the start of the program (line 11), then for each k-mer we find, we will look up the current count for it in the dictionary (line 19). If the k-mer is not found in the dictionary (i.e. this is the first time we've seen that particular k-mer) then we will say that the current count is zero. We'll then add one to the current count (line 20) and store the result back in the dictionary (line 21).

```
import os
kmer_size = 6

def split_dna(dna, kmer_size):
    kmers = []
    for start in range(0,len(dna)-(kmer_size-1),1):
        kmer = dna[start:start+kmer_size]
        kmers.append(kmer)
    return kmers

kmer_counts = {}
for file_name in os.listdir("."):
    if file_name.endswith(".dna"):
        print("reading sequences from " + file_name)
        dna_file = open(file_name)
        for line in dna_file:
            dna = line.rstrip("\n")
            for kmer in split_dna(dna, kmer_size):
                current_count = kmer_counts.get(kmer, 0)
                new_count = current_count + 1
                kmer_counts[kmer] = new_count

print(kmer_counts)
```

This program generates a lot of output! Here are the first few lines so we can see that it's working:

```
{'gcagag': 11, 'aaataa': 13, 'ctttag': 11, 'gcagac': 14, 'ctttaa': 12,
'gcagaa': 15 etc. etc.
```

As planned, we end up with a big dictionary where the keys are kmers and the values are their counts.

Next, we have to process the `kmer_counts` dictionary. We'll go through the items in a loop, and if the count is greater than some cutoff, we'll print the count. For testing, we'll fix the cutoff at 23 (later on we'll make this a command-line option). Here's the code to process the dictionary:

```
count_cutoff = 23
for kmer, count in kmer_counts.items():
    if count > count_cutoff:
        print(kmer + " : " + str(count))
```

And here's the output we get:

```
agagat : 26
agcggg : 26
atcgga : 25
aaggag : 25
cccagc : 24
aggttc : 25
agatta : 24
tctagg : 24
gagtgg : 28
ccggtt : 26
gagcag : 24
ttctga : 26
agatgg : 24
tctgaa : 24
gcgggt : 25
ttcaaa : 25
gattaa : 25
ccagcg : 25
ggacgt : 27
atggct : 24
```

Nearly done. The final step is to replace the hard-coded values for the k-mer size and the count cutoff with values read from the command line. We just have to import the `sys` module, and convert the arguments to numbers using the int function. As specified in the exercise description, the first command line argument is the k-mer size and the second is the cutoff. Here's the final code with comments:

```
import os
import sys

# convert command line arguments to variables
kmer_size = int(sys.argv[1])
count_cutoff = int(sys.argv[2])

# define the function to split dna
def split_dna(dna, kmer_size):
    kmers = []
    for start in range(0,len(dna)-(kmer_size-1),1):
        kmer = dna[start:start+kmer_size]
        kmers.append(kmer)
    return kmers

# create an empty dictionary to hold the counts
kmer_counts = {}

# process each file with the right name
for file_name in os.listdir("."):
    if file_name.endswith(".dna"):
        dna_file = open(file_name)

        # process each DNA sequence in a file
        for line in dna_file:
            dna = line.rstrip("\n")

            # increase the count for each k-mer that we find
            for kmer in split_dna(dna, kmer_size):
                current_count = kmer_counts.get(kmer, 0)
                new_count = current_count + 1
                kmer_counts[kmer] = new_count

# print k-mers whose counts are above the cutoff
for kmer, count in kmer_counts.items():
    if count > count_cutoff:
        print(kmer + " : " + str(count))
```

Now we can specify the k-mer length on the command line when we run the program. With a k-mer length of 6 and a cutoff of 25:

```
 python kmer_counting.py 6 25
```

```
we get the output
agagat : 26
agcggg : 26
gagtgg : 28
ccggtt : 26
ttctga : 26
ggacgt : 27
```

With a k-mer length of 3 and a cutoff of 900:

```
 python kmer_counting.py 3 900
```

```
tct : 908
ttc : 924
gtt : 905
gat : 904
gga : 910
atc : 905
```