Printing and manipulating text: 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 As, the number of Ts, 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 As and Ts 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):

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

The output from this code looks vaguely right:

```
TCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCATCG
ATCGATATCGATGCATCGACTACTAT
```

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. Let's try again:

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

```
my_dna =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGA
TCGATCGATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:63]
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:

```
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCATC
GATCGATATCGATGCATCGACTACTAT
```

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 =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGA
TCGATCGATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:63]
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 =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGA
TCGATCGATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:63]
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 =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGA
TCGATCGATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
exon1 = my_dna[0:63]
intron = my_dna[63:90]
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:

```
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGatcgat
cgatcgatcgatcgatcatgctATCATCGATCGATATCGATGCATCGACTACTAT
```

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[63:90].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[0:63] + my_dna[63:90].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.

Reading and writing files: 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 =
"ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGA
TCGATCGATCGATCGATCATGCTATCATCGATCGATATCGATGCATCGACTACTAT"
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`:

```
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('-', ''))
```

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

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')
```

There's an exercise that uses different techniques to solve a very similar problem at the end of the chapter on functional programming in *Advanced Python for Biologists* – if you find yourself carrying out this type of process in real life code, then it's probably worth a look.

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.

Lists and loops: 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 :
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCGCGATCGATCGATATC
GATCGATATCATCGATGCATCGATCATCGATCGATCGATCGATCGA
coding sequence is :
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCGCGATCGATCGATATC
GATCGATATCATCGATGCATCGATCATCGATCGATCGATCGATCGACGATCGATCGATCGTAGCTAGC
TAGCTAGATCGATCATCATCGTAGCTAGCTCGACTAGCTACGTACGATCGATGCATCGATCGTA
coding sequence is :
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCGATCGCGATCGATCGATATC
GATCGATATCATCGATGCATCGATCATCGATCGATCGATCGATCGACGATCGATCGATCGTAGCTAGC
TAGCTAGATCGATCATCATCGTAGCTAGCTCGACTAGCTACGTACGATCGATGCATCGATCGTACGAT
CGATCGATCGATCGATCGATCGATCGATCGATCGATCGTAGCTAGCTACGATCG
```

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()
```

Writing our own functions: 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.

Conditional tests: 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. We must be careful to convert the fourth column – the expression level – into an integer so we can compare it:

```
# 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 = int(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
```

This general type of problem is very common in programming. There's a similar exercise at the end of the chapter on functional programming in *Advanced Python for Biologists* which illustrates a different approach to solving them.

Regular expressions: 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))
```

Dictionaries: 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 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

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.

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:

```
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:

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.

```
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"
```

Files, programs, and user input: 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
```