Chapter 11

MERGING FILES

Many instruments and archives store data into a series of files. Conversely, when it comes time to analyze data, you will often want to combine data from several files into a single file. In some cases this is as simple as joining the files end to end, but usually it requires combining the data in more complex ways. In this chapter, you will learn to read data from multiple files, combine the data in new ways, and then write the data to a file in a different format. You will also learn how to allow the user to specify filenames, folders, and other parameters when your program is run from the command line.

Reading from more than one file

In Chapter 5 you saw how to use the shell commands `cat` and `grep` to join several files into one long file. This is great for a multifile time series or other datasets where the values can be placed end to end with minimal modification. However, it is more difficult to add values side by side. For instance, you will often have several data files that have lines in X Y format where the X values of all the files are the same. Analysis may require rearranging and combining the data to form a single file with lines in an X Y Y Y format. For example, X could be a specimen number and the Y values could be different measurements for each specimen or X could indicate different measurements for a particular specimen and Y the value of those measurements in different experiments.

In the next example, you will see how to generate a list of all the files of a certain type within a directory, then extract values from those files and combine them into a single master file. Figure 11.1, later in this chapter, will illustrate this process visually. The example will also demonstrate several new Python capabilities. These include taking user input directly from the command line with `sys.argv`, opening several files in succession, building a list, and printing status messages using `sys.stderr.write()`. Otherwise the program is very similar to the file-reading program shown in Chapter 10.

In the terminal, change directories via `cd  ~/pcfb/examples/spectra/` to the example folder we will be using. The directory contains a series of emission spectra (intensity at a certain wavelength) for various colors of light-emitting diodes:

```
host:spectra lucy$ ls
LEDBlue.txt    LEDGreen.txt    LEDRed.txt    LEDYellow.txt
host:spectra lucy$ less LEDBlue.txt
x       BlueLED
350.12  4
350.48  8
350.85  3
351.21  11
351.58  13
351.94  12

...
850.06  10
850.36  6
850.67  7
850.97  10
(END)
```

You can get a quick overview of the beginnings and ends of all the files using the shell commands `tail LED*` and `head LED*`. Notice that the values of `X` in the first column are the same for all the files, since these data were collected using the same spectrometer. Not only are the values the same in all the files, but they occur in the same order.

Now that you know the overall structure of the input files, you can start coding. Switch to a text editor, create a new text file, and save it as `filestoXYYY.py` in your `~/scripts` folder. Although it is not yet a script, you can make it executable with the usual command:

```
host:spectra lucy$ chmod u+x ~/scripts/filestoXYYY.py
```

Because this command specifies the path of the script file relative to your home directory, it will work without your needing to move out of the `examples/spectra` folder.

Getting user input with `sys.argv`

The first component of this program will be a mechanism for getting user input by passing arguments when the program is launched. You have already been using command-line arguments with other programs, such as `ls`. For example:

```
ls -l LED*
```

passes the arguments `-l` and `LED*` to the `ls` command.

In Python, command-line arguments are accessed through the `sys.argv` variable, which is provided via the `sys` module. To see how this works, enter these lines into your blank `filestoXYYY.py` script:

```python
#!/usr/bin/env python
import sys
for MyArg in sys.argv:
    print MyArg
```

Once you import the `sys` module in your script, you have access to the `sys.argv` variable. This variable is a Python list of arguments sent to a program when the user executes it. It provides a way to pass data from the command-line world to the Python world.

Save the file and run it in the terminal window:

```
host:spectra lucy$ filestoXYYY.py
/Users/lucy/scripts/filestoXYYY.py
```

Interesting! Even though you didn't pass any arguments to `filestoXYYY.py`, the `sys.argv` variable was not empty. This is because the zeroth element of the `sys.argv` list is the name of the script being run. This is important to remember when you use arguments in Python. In other words, don't assume that the first argument you have passed to the program on the command line is the first argument in the list. Try running `filestoXYYY.py` again, and add some arguments of your own:

```
host:spectra lucy$ filestoXYYY.py first second "third and fourth"
/Users/lucy/scripts/filestoXYYY.py
first
second
third and fourth
```

As you can see, each argument was separated by a space on the command line, and each was brought into the program as a string. If you want to include spaces as part of an argument, therefore, you need to either surround it with quotes, or escape each space with a backslash. It's also important to know that if you redirect the output of the program to a file using >, the command-line text from > onward is not included in the arguments.

So far this is pretty straightforward. Now try:

```
host:spectra lucy$ filestoXYYY.py LED*.txt
/Users/lucy/scripts/filestoXYYY.py
LEDBlue.txt
LEDGreen.txt
LEDRed.txt
LEDYellow.txt
```

Perhaps this was not the output you were expecting? (If you don't get similar output, make sure you are in your spectra folder.) In this case, the shell itself is expanding LED*.txt to generate a list of matching filenames. This expanded list of files is then sent to the Python script as individual arguments. The shell is acting as a middleman between the user and script. You can get this same list using the shell's echo command:

```
echo LED*.txt
```

Notice that the file list is returned in alphabetical order here. If you were to run this from a different directory and no files matched the string, then you would just get the string itself, with the asterisk, as a normal text argument.

Converting arguments to a file list

Here you will take advantage of the shell's ability to give us a file list, and you will operate on each of these files. First, add a bit of user interface robustness to your program by checking to make sure an argument has been given. If there is only one argument—the name of the program, which is always the first (that is, zeroth) element of the sys.argv list—then you will print out some text describing how the program should be used. Otherwise, use the argument list as a list of files and continue. Your program should look something like this:

```python
#!/usr/bin/env python

Usage = """
filestoXYYY.py - version 1.0
Example file for PCfB Chapter 11
Convert a series of X Y tab-delimited files
to X Y Y Y format and print them to the screen.
Usage:
  filestoXYYY.py *.txt > combinedfile.dat
"""

import sys

if len(sys.argv)<2:
    print Usage
else:
    FileList= sys.argv[1:]
    for InfileName in FileList:
        print InfileName
```

First, try running this program without any arguments. Next, run it with ~/scripts/*.py or *.txt as an argument, or even with a list of full filenames separated by spaces: LEDBlue.txt LEDRed.txt. Remember that the additional information sent to the file begins at the second element of sys.argv. When coding, you can access the arguments from this second element to the end of the list by using the index number followed by a colon alone:

> All arguments passed to your program by **sys.argv**, even numbers, arrive as strings. If the information entered by the user needs to be used as an integer or floating point value, it will have to be converted first using the function **int()** or **float()**.

```
sys.argv[1:]
```

Providing feedback with sys.stderr.write()

This example program, like many others you will write, is designed to be used with the redirect operator > rather than by opening and writing to a destination file. The advantage of this approach is that it gives the user the ability to see the output, make sure it is what was expected, and then choose an appropriate filename in which to store the result. However, a problem with this approach is that any feedback to the user—including progress reports and other information that is useful when the program is running, but not useful when analyzing results—will also be captured in the file. This can complicate later analyses, since such information would likely need to be removed before further work with the file.[1]

Fortunately, there is a way to open up a separate pipeline of communication from the program. You have been using the system's standard input/output, known as stdio, each time you display something on the screen with a print statement. This output is what is captured during redirection with > or >>. You can also send output to the screen through a different pipe, used to report errors and warnings, known as stderr. This is writable as though you were writing to a file:

```
sys.stderr.write("Processing file %s\n" % (InfileName))
```

The sys.stderr.write() statement, like the file-related .write() statement, takes a string as a parameter and writes it as-is. If you want a line ending you have to add it yourself. You can build up a string from multiple pieces of data using the formatting operator % to convert and insert values, as shown here, or through string addition—that is, joining strings with +. Unlike the print statement, you can't mix integers and strings without converting the integers with the str() function or % operator.

---

[1] The two authors disagree as to whether this complication means you should provide more feedback or less when writing a program. One of us feels that this is an incentive to minimize feedback, while the other feels it is best to provide as much feedback as possible, using the sys.stderr.write() mechanism described in this section. We both agree that as a beginning programmer, you should select the approach which best fits your own personal preference.

Now substitute the `sys.stderr.write()` command in place of the `print` statement in your loop, where it currently prints `InfileName`. The output will still appear on the screen, but this way it will not be written to a file if you redirect the output with >. This is a good way for the program to provide feedback on its status, without causing you to go back and comment out these diagnostic statements when you are *really* ready to use it.

RETRIEVING THE CONTENTS OF A DIRECTORY

Although it is often desirable to ask the user which files they want to analyze, there will be times when you want to retrieve a directory list in your program without user input. Two convenient functions can do this. To get a full listing of a particular path, you can `import os` and use `os.listdir()`. This function takes a path in the format of `'/Users/lucy/Documents/'` as a parameter. If the directory does not exist, an error is returned. To do a wildcard search, you can `import glob`[a] and use `glob.glob()`[b] with a path in the format `'/Users/lucy/Documents/*.txt'` as the parameter. If the directory does not exist, an empty list is returned rather than an error.

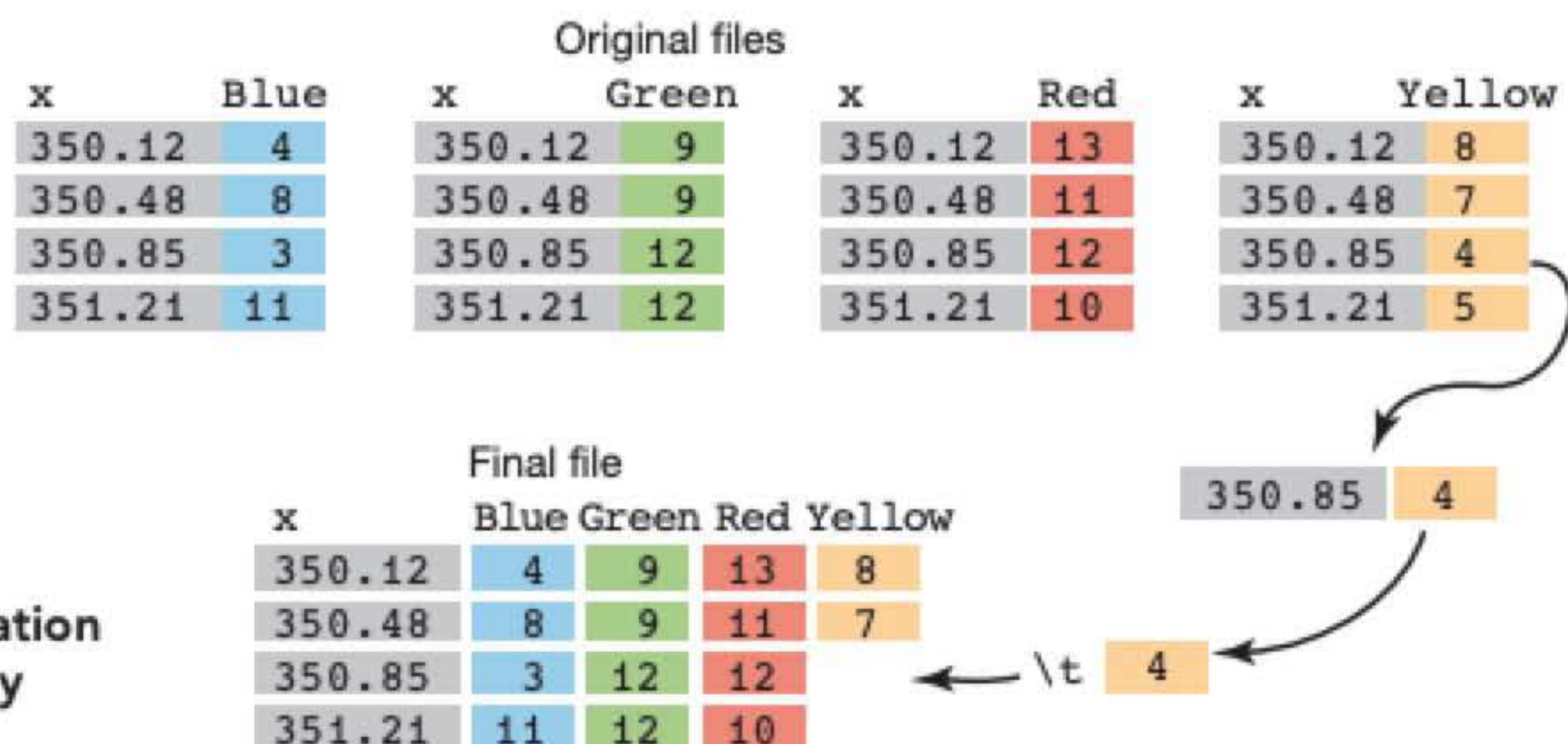[a] We are not making this up.
[b] Don't look at us.

Looping through the file list

Now you have a `for` loop that cycles through the filenames and prints each one. You can now insert code for file opening, reading, and output into this loop.

Now take stock of everything the program needs to do with each file (Figure 11.1). It should open the file and read in the lines, skipping the header line. If this is the first file, you will create a list variable containing the X and Y values together as a text string. For each subsequent file, you will extract the Y value from each line, then append it to the corresponding line of text from the first file, separated by a tab. The table is built column by column, value by value. Once this master data list has been constructed, you will print out the aggregated lines, one by one. Again, for this program to do something meaningful, each file must have data that correspond line by line.

The code to accomplish the first part of this task, building the master list, should be indented within the file-listing `for` loop you have created. (The beginning of

FIGURE 11.1 Graphic representation of the data reorganization done by `filestoXYYY.py`

Original files

x	Blue	x	Green	x	Red	x	Yellow
350.12	4	350.12	9	350.12	13	350.12	8
350.48	8	350.48	9	350.48	11	350.48	7
350.85	3	350.85	12	350.85	12	350.85	4
351.21	11	351.21	12	351.21	10	351.21	5

Final file

x	Blue	Green	Red	Yellow
350.12	4	9	13	8
350.48	8	9	11	7
350.85	3	12	12	
351.21	11	12	10	

350.85   4

\t   4

the program is not repeated here, but the additional portion of your code should look something like this. It can also be seen in its entirety later in this chapter and as `filestoXYYY.py` in the example scripts folder.)

```python
MasterList = []
for InfileName in FileList: # statements done once per file
    Infile = open(InfileName, 'r')
    # the line number within each file, resets for each file
    LineNumber = 0
    RecordNum = 0   # the record number within the table

    for Line in Infile:
        if LineNumber > 0:   # skip first line
            Line=Line.strip('\n')
            if FileNum == 0:   # first file only, save both x & y
                MasterList.append(Line)
            else:
                ElementList = Line.split('\t')
                MasterList[RecordNum] += ( '\t' + ElementList[1] )
                RecordNum += 1
        LineNumber += 1
    Infile.close()
    FileNum += 1 # the last statement in the file loop
```

The important thing to understand here is what happens to `MasterList` while reading the first file versus what happens when reading subsequent files.

Before opening any files, `MasterList` is given the initial value of an empty list, as indicated by the brackets [ ]. This tells the program that the variable is going to be treated as a list of some sort. The first time through the file loop, each line that is read in is appended to the end of `MasterList`, creating a list of strings. After the first five data lines, the list would look like this:

```
['350.12\t4',
 '350.48\t8',
 '350.85\t3',   ← Think of it as two columns of tab-separated values
 '351.21\t11',
 '351.58\t13']
```

With later files, `FileNum` is greater than 0, so the code under the `else` statement is executed instead. First the line which has been read from the file is split at each tab, and put into the variable `ElementList`:

```
['350.12','9']
```

At this point, the X value for each row has already been stored in `MasterList`, so you only want to grab the second element, which has an index of 1. Add a tab

before this item and add it onto the corresponding line of `MasterList`, which has already been defined the first time through the loop, to give:

```
'350.12\t4\t9'
```

To keep track of what record in `MasterList` corresponds to the line currently being parsed, use the `RecordNum` variable, which is reset for each file and increments each time through the loop. This is used as an index to indicate the appropriate record, based on the list that was built the first time through the loop.

Before beginning to read in values from the third file, `RecordNum` is reset to 0 so that it starts over at the first item in `MasterList`. Once the file loop is completed, the items in `MasterList` will look like this:

```
['350.12\t4\t9\t13\t8',
 '350.48\t8\t9\t11\t7',
 '350.85\t3\t12\t12\t4',
 '351.21\t11\t12\t10\t5',
 '351.58\t13\t7\t14\t8',
 ...
```

To see this (long) list, you could add a `print MasterList` statement at the end.

Printing the output and generating a header line

Once the files have been processed, generating output is simply a matter of looping through the items in `MasterList`:

```
for Item in MasterList:
    print Item
```

This code is indented to the same level as the file-reading `for` loop, so that it remains inside the first `else` statement—meaning that the user entered more than 0 arguments upon execution.

One bit of information lost during this process is which spectrum came from which file. An easy way to preserve this is to use the filenames themselves to label the top of the Y columns. This variable, which you'll call `Header`, should be pre-initialized with the name of value in the X column, in this case `'lambda'` (for the wavelength). Each time through the file-reading loop, you can add `\t` plus the filename (stored in the variable `InfileName`) to the `Header` string:

```
Header += '\t' + InfileName
```

Then, at the end of the program, just before you print the `MasterList`, you can print the `Header`. You could also consider stripping off any text after the base filename, for example " `.txt` ", using the `re.sub()` function.

Avoiding hardcoded software

Scripts usually start out working well for a certain task and then get repurposed for other tasks. In our case, the script works fine if there is one header line in the files, but it will fail or generate contaminated data if there are additional lines. The number of header lines in this case is indicated only in one place in the program:

```
if LineNumber > 0:  # skip first line
```

If you wanted to reuse this program for a file that had 17 header lines (it happens) then you would have to open this program, figure out how it works again, and determine a number to put into this `if` statement ("Let's see, 18—no, 16!"). The situation is even worse when a value is used at several places in the program, meaning you need to look at every statement with a 0 throughout the program to determine if it is doing something based on the header lines.

For such reasons, it is often best to create an extra variable that is used anywhere there is a decision involving a value that may be modified later. In this case, you will define `LinesToSkip` = 1 near the beginning of the script, and then change the `if` statement to read:

```
if LineNumber > (LinesToSkip - 1):
```

Now when you return to this script with data files of a different origin, you just have to change the variable definition at the beginning, and you will know that the rest of the occurrences of this variable should work without a problem.

Using the same type of construction, you could add a bit of code allowing the user to enter the number of lines to skip as the first argument, before the list of filenames:

```
filestoXYYY.py 17 *.txt
```

We won't add this option here, but take a moment to consider how you would implement this, including what other statements would need to be modified for your new program to work (not the least of which is the `Usage` string).

With a few modifications, the full code for reading files and converting data in X Y format to X Y Y Y is summarized here. It is also available as a program file in the examples folder, `filestoXYYY.py`. Use this program as a starting point when you need to read in data from multiple files, even if the exact transformation you need is not the same.

```python
#!/usr/bin/env python

Usage = """
filestoXYYY.py - version 1.0
Convert a series of X Y tab-delimited files
to X Y Y Y format and print them to the screen.
Usage:
  filestoXYYY.py *.txt > combinedfile.dat
"""
import sys

if len(sys.argv)<2:
    print Usage
else:
    FileList= sys.argv[1:]
    for InfileName in FileList:
        print InfileName
    Header = 'lambda'
    LinesToSkip=1

    # change this for comma-delimited files
    Delimiter='\t'
    MasterList=[]
    FileNum=0
    for InfileName in FileList:
        # use the name of the file as the column Header
        Header += "\t" + InfileName
        Infile = open(InfileName, 'r')
        LineNumber = 0 # reset for each file
        RecordNum = 0

        for Line in Infile:
            # skip first Line and blanks
            if LineNumber > (LinesToSkip-1) and len(Line)>3:
                Line=Line.strip('\n')
                if FileNum==0:
                    MasterList.append(Line)
                else:
                 ElementList=Line.split(Delimiter)
                    if len(ElementList)>1:
                        MasterList[RecordNum] += "\t" + ElementList[1]
                        RecordNum+=1
                    else:
                     sys.stderr.write("Line %d not XY format in file %s\n" \
                     % (LineNumber,InfileName))
            LineNumber += 1 # the last statement in the Line/Infile loop

        FileNum += 1
        Infile.close() # the last statement in the file loop
```

```
    # output the results
    # these are indented one level to stay within the first "else:"
    print Header
    for Item in MasterList:
        print Item

    sys.stderr.write("Converted %d file(s)\n" % FileNum)
```

If you try to apply this program to other data files, but it does not seem to read the lines properly, try modifying the statement for stripping end-of-line characters by changing it to the following, which will also strip off the alternative end-of-line character \r:

```
    Line=Line.strip('\n').strip('\r')
```

Another solution is to open the file using the parameter 'rU' instead of 'r'.

```
    Infile = open(InfileName, 'rU')
```

This will recognize either line-ending character (\n or \r) as it reads in the lines and will make your program compatible with Unix or Windows line endings.

Other applications of file reading

As discussed at the beginning of Chapter 10, data files are often presented with a single data element spanning several lines within the file. One such format is a molecular sequence format called FASTA. Files with this format put the sequence name of each record on a line beginning with >, with the subsequent line or lines containing the corresponding DNA or amino acid sequence:

```
>Avictoria
MSKGEELFTGVVPILVELDGDVNGQKFSVRGEGEGDATYGKLTLKFICT---TGKLPVP
WPTLVTTFSYGVQCFSRYPDHMKQHDFLKSA---M-PEGYVQERTIFY--------KD
>Pontella
MPDMKLECHISGTMNGEEFELIGSGDGNTDQGRMTNNMKSI---KGPLSFSPYLLSHIL
GYGYYHFATFPAGYE--NIYLHA---MKNGGYSNVRTERY--------EDGGIISITF
```

You have a couple of options for reading files of this type: you can read the names into one list and the sequences into another, or you can load the names as keys in a dictionary with the sequences as corresponding values. The dictionary approach makes it easy to access a particular sequence within your program, but because dictionaries cannot have repeated keys, it won't be able to handle situations where two sequences have identical names. A possible solution demonstrating both approaches is shown here and is available in the example scripts as `seqread.py`:

```python
#!/usr/bin/env python

Usage="""
seqread.py - version 1
Reads in a file in fasta format into a list
and a dictionary.

The resulting list is formatted
[['name1','sequence1sequence1sequence1'],
 ['name2','sequence2sequence2sequence2']]

Usage:
  seqread.py sequence.fta"""

import sys

# Expects a filename as the argument
if len(sys.argv) < 2:
    print Usage

else:
    InfileName= sys.argv[1]
    Infile = open(InfileName,'r')
    RecordNum = -1  # don't have the zeroth record yet

    # Set up a blank list and blank dictionary
    Sequences=[]
    SeqDict={}

    for Line in Infile:
        Line = Line.strip()
        if Line[0]=='>': # we have a new record name
            Name=Line[1:]  # chop off the > at the front

            # Make a two-item list with the name as the first element,
            # and an empty string as the second

            Sequences.append([Name,''])
            RecordNum += 1 # Now we have a record

            # Use the Name for the dictionary key
            SeqKey = Name
             # create a blank dictionary entry to append later
            SeqDict[SeqKey] = ''

        else:  # this means we are not on a line with a name
            if RecordNum > -1:  # are we past any header lines?
                # Add on to the end of the 2nd element of the list
                Sequences[RecordNum][1] += Line
                # Add to the dictionary value for the present key
                SeqDict[SeqKey] += Line
```

```
# when done with the loop, print the sequences:
# insert your processing and file output commands here
for Seq in Sequences:
    print Seq[0],":", Seq[1]

# could also print a list of all the names (=keys)
print SeqDict.keys()
```

If you turn back to look at Figure 10.1 you will see that this program fulfills the task shown in examples B and C of that figure: a few lines of input are used to create corresponding data series, either as parallel lists or as dictionary keys and values. Both methods are employed in this program, allowing you to choose whichever is more appropriate to your needs. In this case, we have made a two-dimensional list—basically two columns of data—where the first column is the names, and the second column is the sequences. To use the data, you would create a file-processing loop that cycled through each value in the list and output your desired subset of sequences or sequence-based calculations.

SUMMARY

You have learned how to:

- Gather user input using `sys.argv`

- Process several files in sequence using a `for` loop

- Use variables instead of fixed or hardcoded values in your programs

- Give warnings and feedback using `sys.stderr.write()`

Moving forward

- For some programs that take arguments, it is useful to have a default value which is used when no arguments are provided by the user. Add a `DefaultValue` variable to `filestoXYYY.py` along with the necessary logic to make it operate in this way.

- If you add a default value, as in the previous exercise, you lose the ability to print your `Usage` description automatically for naïve users. Think of ways to solve this. One way might be to check to see if the user has specifically asked for help; if so, you could see if `sys.argv[1]=='help'`. Another way might be check whether the arguments entered by the user

seem to make sense, and to provide help if the check fails. For example, if the first argument should be a filename, you could check if such a file exists, using `os.path.isfile()` in the os module. If the argument should be a number, you could check it via the built-in string method `.isdigit()`.

- Add more error checking to `filestoXYYY.py`, to allow for blank lines and for lines which don't parse correctly with the `.split()` function. You can do this at the statement that checks the header lines, or at the point where the line is split into elements.

- Modify the program `filestoXYYY.py` to rearrange the columns of a file, by outputting the fields of `MasterList` in a different order.