Chapter 10

READING AND WRITING FILES

The programs you have created so far rely on information being written into the program itself, or else provided by the user at a prompt. In most cases, though, you will want to process data that are stored in files. In this chapter you will learn how to open text files, parse the data, and then use that information to generate new text files. You will develop these skills by building a file converter program that reads in a tab-delimited file containing latitudes and longitudes, then writes out a file that can be visualized with **Google Earth**. In the course of building up this example, you will get experience in Python with file handling, regular expressions, several new functions, and a variety of other tools.

Surveying the goal

This chapter will focus on building up a program that can read location data from one text file format, convert it to another format, then rewrite it in another file. Specifically, this program will reformat an input file with a series of locations to create an output file that can be read and displayed by the geographical viewer Google Earth. This is a very typical challenge for biologists—data are in hand and you know what you want to do with them, but the format of the data isn't understood by the program you want to use.

The input file, `Marrus_claudanielis.txt`, contains the latitude, longitude, depth, and associated data for several specimens that were considered when a new species of siphonophore, *Marrus claudanielis*, was described. The first challenge is to simply read the text from the file. Next, the individual components of the data must be parsed—that is, extracted from each line. Finally, these data must be repackaged into a new format and written to an output file.

Before you start writing code for any project, it is important to take stock of what you have and where you want to go, and then map out a clear strategy.[1] In particular, you want to try to anticipate any complex or problematic steps in the process. Even though such issues may seem specific to a small part of the task, it is not uncommon for a few sticking points to have a big effect on the overall structure of a program. For example, how are the data separated? Were they generated automatically (thus, likely to be consistent) or hand-entered by a person (and therefore likely to require error-checking)? If you don't take these issues under consideration from the very start, you may find yourself needing to rewrite parts of your program as you go, or even reversing direction and adopting a different strategy. At worst, you could introduce errors into your analysis. Once you have a good sense of all the steps that are required, you can break the problem down into smaller goals that you can work on in pieces.

The first step is to see what data you have. Open the example file `Marrus_claudanielis.txt` in your text editor and get a feel for how the file is structured:

```
Dive∆          Date∆       Lat∆          Lon∆           Depth∆  Notes
Tiburon 596∆   19-Jul-03∆  36 36.12 N∆   122 22.48 W∆   1190∆   holotype
JSL II 1411∆   16-Sep-86∆  39 56.4 N∆    70 14.3 W∆     518∆    paratype
JSL II 930∆    18-Aug-84∆  40 05.03 N∆   69 03.01 W∆    686∆    Youngbluth (1989)
```

The data are organized one specimen per row, beginning with a header row that describes which data can be found in each column. There is a tab between each data field (shown by the ∆ symbol, which you can reveal in TextWrangler using Show Invisibles), and some fields can contain spaces. The latitude and longitude are each in their own column. These coordinates are formatted in degrees and minutes (the minutes can have fractions), with a letter indicating North, South, East, or West. There is a single space between the degrees, minutes, and compass letter. The Notes field at the end of the line can also contain spaces, or it can be blank.

Now take stock of where you want to go. The intermediate step will be to split the values from the columns. The ultimate goal is to write a file that Google Earth can load, formatted in Keyhole Markup Language, or KML. There are a couple of ways to figure out file formats. If you are lucky, the file format will have a good technical definition that explains all its parts and ensures that there is no confusion across programs or among programmers. In this case, Google's KML file format has a thorough definition (see `code.google.com/apis/kml/documentation/`), but more often than not, you will be faced with file formats that have never been properly specified. In those cases you must take a look at an existing file and reverse-engineer its structure.

[1] This is equivalent to creating an outline before you write a paper. We may not always operate this way, but the results, especially in a more complex paper, are usually better when we do.

Here, the example file `Marrus_claudanielis.kml` is what you are hoping to produce at the end of the process, and so it will give you a sense of how a KML file is organized. To take a look at its structure, open it in a text editor rather than in Google Earth. At this point, don't worry about all the formatting characters; we will get to that stage later. For now, just identify the bits of data that will need to be generated from the input file and figure out how these bits will need to be modified, going from one file type to the other. A quick examination indicates that the position coordinates need to be in decimal degree format, rather than in degrees and minutes, with a minus sign rather than letters to indicate South and West. The depths also need to be in negative numbers, reflecting that they are below sea level.

Now you know some specifics about your starting and ending points. Rather than take on the entire transformation at once, you will first focus on extracting the data out of the input, in the process converting the geographic coordinates from degrees and minutes to decimal degrees. You will then write these converted data to a file, though not yet in KML format—that can wait until you have taken care of all other data processing that needs to be done.

Reading lines from a file

Considerations before reading a data file

As in most programming languages, there is more than one way to read files in Python. In particular, you can read the whole file into memory at once, or step through it line-by-line (Figure 10.1). Both methods involve creating a variable that represents the stream of data contained in the file.

When you are considering how to read your dataset, there are two main questions to ask: First, is each data value contained within a single line (as is the case with `Marrus_claudanielis.txt`, in which each line contains information on one specimen), or can a data value continue across lines (as with molecular sequence data, where one sequence can be split across several lines)? Second, does the output depend only on the values contained in a single line (such as when converting units, or generating a new value based on other values in that line) or does it depend on knowing values from other lines in the file (such as a sorted list or a running average)?

The simplest case is when there is a one-to-one correspondence between input and output lines, as in file type A in Figure 10.1. Here, no data are combined across lines, and the input and output lines are in the same order. In this case, file processing can generally be handled with a single `for` loop. Strategies for laying out your data are discussed in Chapter 15, but in general you want to strive from the outset—before you even begin your experiment—to organize your records so that associated information is not spread across several lines.

On the other hand, a common task in biological computing is to take a file that is in a non-standard format, and reorganize it into a simple table that can be processed by MATLAB, R, or a spreadsheet program.

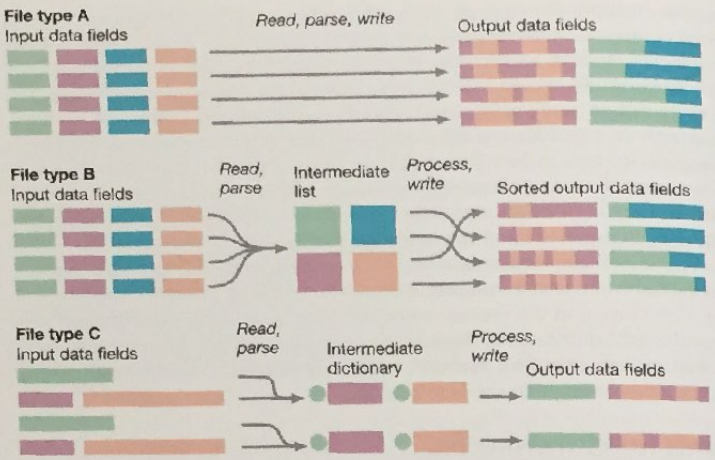

FIGURE 10.1 Processing data within and across lines of a file Parcels of similar data are represented by colored blocks in three different file types. In file type A, each line of output is generated from data within only one line of input, and the order of the lines is not changed. No data need to be stored or reorganized between lines, and each line can be processed in the order it occurs. In file type B, the order or content of data in the output may depend on all input data (such as when the output data are sorted across lines), or the values in the output may depend on a value derived from combined input (such as when all lines are normalized by the average). All input lines must be parsed into internal variables, such as lists, before any data can be processed and written to the output. Each of these input/output processes typically requires its own loop. In file type C, data from two or more lines of input are combined into each output line, such as when a specimen name occurs above a block of data and needs to be incorporated into each output line. For each line of output, data from multiple lines of input must be read, parsed, stored, and processed.

When reading and writing files where there is not a one-to-one correspondence between the lines of the "before" and "after" files, as in file types B and C in Figure 10.1, it is often useful to employ two loops: the first loop reads the data from the input file and stores it in internal variables, and the second loop processes the data and writes the new file. The file-loading loop may also be used to import data from several files into a combined data variable, with the second loop serving to output the aggregated data set; you will explore this approach in the next chapter. Once you create a program to do such operations on one type of file, it is relatively quick to modify the program to handle many other data formats.

Opening and reading a text file

To begin, this simple program will open a text file, read each line one by one, display each line to the screen as it is read, and then close the file:

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

# Set the input file name
# (The program must be run from within the directory
# that contains this data file)
InFileName = "Marrus_claudanielis.txt"

# Open the input file for reading
InFile = open(InFileName, 'r')

# Initialize the counter used to keep track of line numbers
LineNumber = 0

# Loop through each line in the file
for Line in InFile:
    # Remove the line-ending characters
    Line = Line.strip('\n')
    # Print the line
    print LineNumber,':', Line

    # Index the counter used to keep track of line numbers
    LineNumber = LineNumber + 1

# After the loop is completed, close the file
InFile.close()
```

Enter the lines as shown into a text file, then save the file in your `~/scripts` directory as `latlon_1.py` (or just copy `latlon_1.py` from the example scripts to `~/scripts`). Make the file executable, then open a terminal window and `cd` to your examples folder, where the data file `Marrus_claudanielis.txt` is located. Execute the new program by typing `latlon_1.py`.

If the program fails, try making sure that the data file is present in your current directory:

```
ls Marrus_claudanielis.txt
```

If all of the lines of the file are displayed by your program as a single line, check the end-of-line characters in your data file to make sure they are appropriate for your system (see Chapter 1). This is a common problem when reading data files, and fortunately it is usually simple to address by opening the file in a text editor, changing the end-of-line character, and saving the file again. You can also try using `'rU'` instead of `'r'` in the program's `open()` statement. There are ways to do batch conversions of end-of-line characters for more than one file at a time; these will be discussed in Chapter 16.

The comments (the text following the # characters) explain most of what is happening, including some new things you haven't seen until now. The built-in function `open()` is called with two parameters, one specifying the name of the file to be opened and the other specifying the file mode; in this case, the mode is `'r'` for `read`. The file-object returned by `open()` is assigned to the variable `InFile`. Note that the variables `InFileName` and `InFile` are different. Whereas `InFileName` is just a string that contains the path of the file to be opened, `InFile` is an object that allows us to interact with the file and its contents in a read-only mode.[2] Mixing up these two kinds of variables is a common mistake for beginning programmers.

In pseudocode, the basic process for reading a file is this:

```
InFile = open(FileName, 'r')    ← Open up a pipeline to the file
for Line in InFile:             ← Loop through the lines one by one
    pull values from each Line
    store them, do calculations, or write output
InFile.close()   ← Close the file object (not the filename)
```

To keep track of where you are in processing the file, you will keep a running count by adding `1` to the variable `LineNumber` each time through the loop. In this case, the count serves to give you an idea of how the program is progressing; more generally, internal markers of this sort can be useful for debugging and locating irregularities.

The `for` loop processes the file one line at a time. At each iteration of the loop, the `for` statement places the next line from the file into the string `Line`, after which the statements within the loop have access to the data values.

Removing line endings with `.strip()`

Lines are separated within files by end-of-line characters. Although these characters are normally invisible to you, they are retained at the end of each line as the line is read in. It is good practice to strip them off, to avoid having them interfere with later analyses. Here, the first step of processing each line is therefore to remove the end-of-line character with `.strip('\n')`. This string method returns a copy of the string it acts on (in the present program, this variable is `Line`), but first strips away the specified character (in this case, `\n`) from either end of the string. If the specified character doesn't occur at the beginning or end of the string, then the new string is identical to the old one. If you don't specify a character for `.strip()`, all white space, including spaces and tabs, is removed from the beginning and the end of the string. If you are going to be reading files from various sources, you can strip both kinds of line endings in succession with `Line.strip('\n').strip('\r')`.

You also need to remove \r for Windows files.

For the moment, the program just prints out the line number, followed by a colon and the line itself. Remember that the `print` function adds an end-of-line character to the text it displays; thus, even though you are stripping this character as each line is read, it is effectively added back when the line is displayed. The final statement in the loop increments the line counter—that is, adds 1 to its value—before returning to the `for` statement to process the next line. After the loop is done, the file is closed with the `.close()` method, which is built into the `InFile` variable created by `open()`. Remember to close `InFile`, not `InFileName`, and to place the close statement outside of the loop, rather than have it indented within the loop.

Note that the `LineNumber` variable starts out with a value of `0` and is then incremented by 1 as the last step of each cycle of the loop. Because `LineNumber` isn't incremented until the *end* of each loop, the first line is labeled as line `0`, the second line as line `1`, etc. We could have started the count at 1 so that the first line was 1 rather than 0, but it is conventional in computer programs to start counters like this at 0. In this case, the starting number wouldn't have made a substantial difference, but in many cases, such as when indexing lists, it is important to start with a 0. It is therefore best to always number items starting with 0, so that there isn't any confusion about which numbering system is in use.

Skipping the header line

As it stands, the program treats each line the same. However, the example file has a header line, and you will want to skip this line when you extract data from the specimen records. Since you are already keeping track of the line numbers, all you need to do now is insert an `if` statement that only considers lines greater than 0. Since the first line is numbered 0, that line is now skipped. Modify your program to add an `if` statement as follows:

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

# Set the input file name
InFileName = 'Marrus_claudanielis.txt'
# Open the input file for reading
InFile = open(InFileName, 'r')
# Initialize the counter used to keep track of line numbers
LineNumber = 0

# Loop through each line in the file
for Line in InFile:
    if LineNumber > 0:    ← To skip the header line, note next few lines are now further indented
        # Remove the line ending characters
        Line = Line.strip('\n')
        # Print the line
        print LineNumber,":",  Line
    LineNumber = LineNumber + 1

# After the loop is completed, close the file
InFile.close()
```

[2] Within the for loop, the `InFile` object that you create acts like a list of lines in the file. You can create an actual list variable containing all the lines of the file using the command `FileList = InFile.readlines()`.

The line that increments the counter, `LineNumber = LineNumber + 1`, is not under control of the `if` statement—its indentation is the same as that of the `if` statement, so it is executed with each cycle of the loop. If it were within the block of code controlled by the `if`, it would never be executed and `LineNumber > 0` would never be `True`. This wouldn't create an infinite loop, since the `for` statement would still process each line of the file, but none of the lines would be analyzed or printed.

Keep practicing this process of file reading so that you become comfortable with it, and when you come up with some code that works well, reuse it for your other programs. The basic process of opening and looping through a file is one of the most powerful uses of a Python program. You can probably already imagine many potential applications.

Parsing data from lines

Splitting a line into data fields

Now that you have access to the data in this file on a line-by-line basis, you are ready to pluck the needed specimen data from each line. This could be done entirely with regular expressions—and in fact, in some cases regular expressions may be the easiest way to go, such as when formatting is inconsistent. However, when character-delimited text files are formatted in a consistent manner, they are usually most easily approached using `.split()`. This method takes a string and splits it according to a delimiter, thereby producing a list of strings—that is, a list of the values or fields occurring between the delimiters. The delimiters themselves are thrown away. In the list, the fields occur in the same order as they did in the original string.

By default, `.split()` uses white space, in the form of both spaces and tabs, as separators for the columns of data. However, you can specify your own delimiter by adding it as a parameter in the parentheses. For example `Line.split('\t')` would split the line into a list of those elements originally separated only by tabs, not by spaces. (Remember from the chapters on regular expressions that `\t` is the character combination used to represent a tab.) It is desirable for now to keep the latitude/longitude values and the comments unsplit, so we will specify `'\t'`.

To see what such a list looks like, add this line to the program, indented before the `print` statement:

```
ElementList = Line.split('\t')
```

and then modify the `print` statement to display this list instead of the line itself:

```
print LineNumber, ":", ElementList
```

Rerun your edited script and see how the output has been subdivided, and how the values occur within square brackets, indicating they are contained in lists:


```
1 : ['Tiburon 596', '19-Jul-03', '36 36.12 N', '122 22.48 W', '1190', 'holotype']
2 : ['JSL II 930', '18-Aug-84', '40 05.03 N', '69 03.01 W', '686', 'Youngbluth (1989)']
3 : ['JSL II 1411', '16-Sep-86', '39 56.4 N', '70 14.3 W', '518', 'paratype']
4 : ['Ventana 1575', '11-Mar-99', '36 42.24 N', '122 02.52 W', '767', '']
```

You can also test what happens if you omit the `'\t'` parameter and use `ElementList.split()`. You can see that the dive, latitude, longitude, and comments are further subdivided into their own list elements, because they contain spaces.

A subtle but important thing to notice about the results is that there are empty single quote marks (`''`) at the end of those lines where the Notes field is empty. When parsing a file, it is important to recognize that there may be empty data fields such as these, and to make sure that your program is not thrown off—for example, by the presence of two tabs in a row, indicating an empty field.

Selecting elements from a list

Now you are able to read in a file of delimited data and split the constituent columns into a list. Lists are ordered arrays of data, indicated in Python by square brackets, `[]`. The first element in a list is numbered 0, the second element is numbered 1, and so on. To access individual elements of the `ElementList` variable, you put the index or range of indices in `[]` after the name.

For example, each time through the loop, `ElementList[2]` (the third element in the list) will contain the latitude value, and `ElementList[3]` (the fourth element in the list) will contain the longitude value. You can test this by modifying or adding a print statement to display the depth, latitude, and longitude by themselves:

```
print ElementList[4], ElementList[2], ElementList[3]
```

When displayed this way, each field is separated by the space that is inserted by `print` when commas are used. To separate the fields with a tab, use the format operator to get more control over the string:

```
print "Depth: %s\tLat: %s\tLon: %s" % (ElementList[4], ElementList[2], \
 ElementList[3])
```

This portion of code also introduces a technique for splitting a long line into multiple lines by putting a backslash (\) at the very end of a line. This continuation character causes the line that follows to be interpreted as though it was on the same line as the backslash. In effect, you are escaping out the end-of-line character. The indentation of the second line does not matter when using the backslash in this way.

By substituting as we have into the `%s` string placeholders, we obtain this output:

```
Depth: 1190    Lat: 36 36.12 N    Lon: 122 22.48 W
Depth: 518     Lat: 39 56.4 N     Lon: 70 14.3 W
Depth: 686     Lat: 40 05.03 N    Lon: 69 03.01 W
```

Remember at this point that these values are strings and have not been converted to integers or floating point values. This is why we use `%s` to plug the strings directly into the output, rather than `%d` or `%f` as we would when formatting numerical values. This version of the program is saved as `latlon_2.py` in your example scripts folder.

Writing to files

The next major skill in working with files is to write your output to a new file. So far you have been printing all your output to the screen. You could capture this text using the shell's redirect operator (>>) as described in Chapter 5, but you will often want to create a file directly from within the program.

Getting set up to write data to a file is very much like getting set up to read from a file, in that you use the `open()` command and a filename:

```
OutFileName = "MarrusProcessed.txt"
OutFile = open(OutFileName,'w')
```

Here you are using the `'w'` parameter to indicate that you are getting ready to write data to the file named in `OutFileName`, overwriting any previously existing file by that name. There is also the `'a'` option, which will **append** (that is, add) to an existing file, or if necessary create such a file if it doesn't already exist. The `'w'` and `'a'` options are equivalent to the > and >> redirect operators in the shell, where the first overwrites files, and the second appends output to a file.

Now that the file is open, you can write text to it with the `.write()` method; this method is built into the file variable, in this case the variable called `OutFile`. For example, to write just the depth string to a file, within the loop you could say: `OutFile.write(ElementList[4]+"\n")`.

Note that you need to append a line ending character, `"\n"`, or else all of your output will be on the same line. This is because unlike `print`, the `.write()` method does not add a line ending by default.

Once you are done writing your output, you will close the file you have written, using the same method that you used to close the file you were reading. Again, be sure to put the `open()` and `.close()` statements *outside* the loop; you don't want to reopen the file each time:

```
OutFile.close()
```

Put these three components together in your script to generate a version, `latlon_3.py`, which writes the depth, latitude, and longitude to a file:


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

# Read in each line of the example file, split it into
# separate components, and write certain output to a separate file

# Set the input file name
InFileName = 'Marrus_claudanielis.txt'

# Open the input file for reading
InFile = open(InFileName, 'r')

# Initialize the counter used to keep track of line numbers
LineNumber = 0

# Open the output file for writing
# Do this *before* the loop, not inside it
OutFileName=InFileName + ".kml"

OutFile=open(OutFileName,'w') # You can append instead with 'a'

# Loop through each line in the file
for Line in InFile:

   # Skip the header, line # 0
   if LineNumber > 0:

      # Remove the line ending characters
      Line=Line.strip('\n')

      # Separate the line into a list of its tab-delimited components
      ElementList=Line.split('\t')

      # Use the % operator to generate a string
      # We can use this for output both to the screen and to a file
      OutputString = "Depth: %s\tLat: %s\t Lon:%s" % \
         (ElementList[4], ElementList[2], ElementList[3])

      # Can still print to the screen then write to a file
      print OutputString

      # Unlike print statements, .write needs a linefeed
      OutFile.write(OutputString+"\n")

   # Index the counter used to keep track of line numbers
   LineNumber = LineNumber + 1

# After the loop is completed, close the files
InFile.close()
OutFile.close()
```

When writing to files, you need to be careful with your filenames and file-related variables. When you open a file for reading, it is not a serious problem to accidentally specify the wrong filename. You will just open a file you weren't expecting, or you will get a "No such file" error. However, when you are *writing* to files, even the act of opening a file will erase its contents! It is often good practice to check to see if a file exists before trying to open it. One way to do this is with the `os.path.exists()` function. This function is part of the `os` module, which you can access by adding `import os` to the top of your program, after the shebang line but before you invoke any of the module's functions. (You will learn more about importing modules shortly.)

Recapping basic file reading and writing

At this point, you can read lines of data from a file, extract individual fields, and write these fields back out to another file. In the present program, `latlon_3.py`, the sequence of events goes like this:

- Open the files to be used for reading and writing with the `open()` function
- Read in a line as part of a `for` loop
- Check to see if the line number is greater than 0
- Split the line into a list of strings according to tabs
- Generate a formatted output string
- Print that string to the screen (this behavior is used for testing or debugging, but otherwise stays turned off)
- Write the string to a file
- Increment the line number
- Loop back to read and process the next line
- When done with the last line, close the input and output files

This general sequence will be used repeatedly in your work. For the most part, the things that will change from one program to another are how the line values are processed and how the output string is formulated. You will now explore variations of both of these aspects—first, to convert the latitude/longitude values to decimals, and second, to eventually write the output as a Google Earth KML file.

Parsing values with regular expressions

You have split a line into individual elements, but some of these elements now require additional parsing. To convert the latitude and longitude values to their decimal equivalents, you have to separate the corresponding text fields into their three components: degrees, minutes, and hemisphere (i.e., N, S, E, or W). There are

a few ways to do this, but here you will use regular expressions, the search and replace tools you learned about in Chapters 2 and 3.

For the next section, we will mainly be showing portions of the program that occur within the `for` loop after the line is read in, and not showing the full body of the code. We will show the full code with all these modifications at the end of this section.

Importing the `re` module

While there is support for regular expressions in Python, these commands aren't available by default. They are bundled into a module that you must first import into your program. Modules are collections of computer code that are packaged together, and the act of importing them into a program makes them available for your use. Modules will be discussed in greater detail in Chapter 12, where you will find information on additional built-in modules that come with the Python language, more options for importing tools from modules, and information on installing third-party modules that don't come with Python by default.

The name of the regular expressions module is `re`. To import this module into your program add the following line below the shebang:

```
import re
```

It is customary to do your imports close to the top of the file. This ensures that the appropriate modules are loaded before you use them, and makes it easier to glance at a program and get a sense of what tools it uses.

You can also import modules with this same command when working at the Python interactive prompt. Once a module is imported, you can take advantage of the `dir()` and `help()` functions, such as `help(re)`, to get a bit of information on what is available in it.

Using regular expressions with the `re` module

There are several functions in the `re` module, including `re.search()` and `re.sub()`. Regular expressions have the following general format in Python:

```
Result = re.search(RegularExpressionString, StringToSearch)
```

In our case, the string to search will be one of the latitude/longitude values in the list named `ElementList`. Review the output of `latlon_2.py` or `latlon_3.py` to see the format of `ElementList[2]` (the first line of which is shown here) and `ElementList[3]`:

```
"36 36.12 N"
```

Now think back to your regular expression skills. In words, you now want to match "one or more digits, a space, one or more digits or decimal points, a space, a letter." Translated to regular expressionese, this becomes:

```
\d+ [\d\.]+ \w
```

You want to capture the three elements independently, so put parentheses around them:

```
(\d+) ([\d\.]+) (\w)
```

To put this into a Python regular expression and search for it within the string `ElementList[2]`, you will create that query as a string:

```
SearchStr = '(\d+) ([\d\.]+) (\w)'
Result = re.search(SearchStr,ElementList[2])
```

For clarity, we have defined the search string as a variable, `SearchStr`, and then used the variable for the search. However, you could also put the search string directly within the `re.search()` statement where it currently says `SearchStr`.

PYTHON SEARCH STRINGS When you construct search terms in Python for regular expressions, you can generally use the same wildcards and symbols as in TextWrangler. However, some issues arise with backslashes. This is because Python uses backslashes in the same way regular expressions do—to escape special characters. This means that escaped characters are escaped by Python when the search string is defined, before being passed to the regular expression. If the escape sequence is one that is interpreted the same way by both Python and regular expressions, this will have no effect on the result. The two characters `\t`, for instance, will be replaced with an actual tab character by Python before being passed to the regular expression. This tab will match tabs in the string that is being searched, just as the original escape sequence `\t` would have. Likewise, no problematic issues arise when the escape sequence is specific to regular expressions. The two characters `\d`, for instance, don't mean anything to Python, and are passed as-is to regular expressions.

However, if you want to search for a backslash (\) with a regular expression, you need to escape it with a backslash. You therefore need two backslashes (\\) in the search term being passed to the regular expression. But Python also escapes backslashes, so if you put two backslashes in the string you write in your Python code, Python will pass only one of them to the regular expression! You therefore need to escape each of the two backslashes with its own backlash. As an example, searching for a single backslash requires an initial search string with four backslashes. The need to write `\\\\` to get `\\` to search for `\` is so nefarious that it has been given its own name, the "backslash plague."

Apart from writing four backslashes, there is another solution to this problem. This is to turn off character escaping in Python by placing an `r` immediately before the quotes that mark a string (`r` stands for "raw"). For example, if you set `s="c:\\"`, Python will escape the backslash; thus the resulting string `s` will contain only one backslash and will have the value `"c:\"`. Defining a raw string with the statement `s=r"c:\\"`, on the other hand, suppresses Python's character escaping, and the string will have the value `"c:\\"`.

When this command is run it generates a variable called `Result`. But how do you access the results themselves? In fact, search results are a special type of Python object, containing information about what matched within the string, where it matched, and the captured text. These results are stored as groups within the variable `Result`. All the results together can be retrieved with `Result.groups()`; they can also be retrieved individually with the `.group()` method applied to the variable (note that there is no `s` at the end of the method name). The zeroth group (that is, the very first group) is the full match of your search expression, while subsequent groups `(1,2,3)` contain the substrings captured by each pair of parentheses. The following commands retrieve the captured groups:

```
DegreeString = Result.group(1)  # this is equivalent to \1 in TextWrangler
MinuteString = Result.group(2)
Compass      = Result.group(3)  ← The spacing here is just to make it more readable
```

Because you are going to work with the first two values as floating point numbers, you will need to convert them with the `float()` function. Since you don't need the strings themselves for anything, you might as well do this conversion at the same time that you retrieve the regular expression results:

```
Degrees = float(Result.group(1))
Minutes = float(Result.group(2))
Compass = Result.group(3).upper() # make sure it is capital too
DecimalDegree = Degrees + Minutes/60

# If the compass direction indicates the coordinate is South or
# West, make the sign of the coordinate negative
if Compass == 'S' or Compass == 'W':
   DecimalDegree = -DecimalDegree
```

This code is a lot to digest at once. You are using a regular expression to extract three substrings, then converting two of these to floating point numbers, and then using the three values in combination to do a couple of simple calculations. If you want to reinforce the ways that regular expressions are built, try creating some strings in the Python interpreter and testing some searches.

Summary of using `re.search()` *and* `re.sub()`

Usage of the `re` module is summarized in the following session at Python's interactive prompt. We haven't walked you through an example of the `re.sub()` function, but its usage is relatively clear from the context:

```
>>> import re
>>> OrigString = "Haeckel,Ernst" ← The string you wish to search
>>> SubFind = r"(\w+),(\w+)" ← The regular expression to search for
>>> Result = re.search(SubFind,OrigString) ← Perform the search, store the results
>>> print Result.groups() ← See what you found
('Haeckel', 'Ernst')  ← The two captured substrings
>>> SubReplace = r"\2\t\1" ← Set up a string for replacement using raw string style
>>> NewString = re.sub(SubFind,SubReplace,OrigString) ← Format for re.sub()
>>> print NewString ← See the substituted string
'Ernst⌂  Haeckel'
```

Before you go on to incorporate the changes using `re` into our current program, we are going to explain one more important programming capability—that of creating your own functions. This will allow you to reuse this one bit of code on both the latitude and longitude values.

Creating custom Python functions with `def`

Some earlier programming examples in this book included nearly identical blocks of code that did the exact same calculations, but on different variables. In the first version of the `dnacalc.py` script, for example, you needed to perform the same operation on all four bases. You just copied and pasted the same code and modified it to act on `A`, `G`, `C`, and `T` in turn.

If you find yourself copying and pasting the same blocks of code within a program, this is a sign that you should pause and think of another way to achieve those repeated steps. There are several problems with reusing code by copying and pasting within a program: it gets confusing; it is a lot of work if the code is frequently reused; if you want to change one thing in the replicated procedure you will need to fix it in many places; and finally, it makes the program unnecessarily long.

Loops are a great way to reuse the same block of code. You used a loop to simplify the `dnacalc.py` program in the last chapter. Now we introduce another tool for reusing the same piece of code repeatedly: a **function**. You have been using functions since your first Python program. For example, consider the built-in `float()` function. You could conceivably write several lines of code that take a string and convert it to a floating point number, then re-copy those lines wherever you need that function. Not only would this approach make your program unwieldy, but if you wanted to change or fix the code involved, you would have to go through and reproduce that same change every where it occurred in your program.

Built-in functions are nice, but even nicer is building your own functions which allow you to extend the language to suit your own needs. You will now create the function `decimalat()` to allow you to reuse the code that you just wrote to convert a raw string of degrees, minutes, and compass direction to a decimal measurement. Without this function, you would need to copy and paste the code to convert both the latitude and longitude strings to floats. Your function will take

a string as input (such as `"36  36.12  N"`), and return a numerical value (in this case, `36.60200`).

Functions are defined using the term `def` (short for **def**inition), followed by the name you give the function, a pair of parentheses containing the names of any parameters (that is, values) to be sent to the function, and finally, a colon. The function's commands are indented beneath the `def` line, and at the conclusion of the definition, any values to be sent back are specified using the `return` statement. Functions must be defined prior to the point in your program where they are used. To build the `decimalat()` function, take the block of statements explained previously and place it below the line which indicates the name of your new function:

```
def decimalat(DegString):  ← DegString is the value sent to the function as input
   # This function requires that the re module is already imported
   # Take a string with format "34 56.78 N" and return decimal degrees
   SearchStr='(\d+) ([\d\.]+) (\w)'
   Result = re.search(SearchStr, DegString)

   # Get the captured character groups, as defined by the parentheses
   # in the regular expression, convert the numbers to floats, and
   # assign them to variables with meaningful names
   Degrees = float(Result.group(1))
   Minutes = float(Result.group(2))
   Compass = Result.group(3).upper() # make sure it is capital too

   # Calculate the decimal degrees
   DecimalDegree = Degrees + Minutes/60

   # If the compass direction indicates the coordinate is South or
   # West, make the sign of the coordinate negative
   if Compass == 'S' or Compass == 'W':
      DecimalDegree = -DecimalDegree

   return DecimalDegree  ← The value of DecimalDegree is the output sent back
# End of the function definition
```

At the end of the indented block which defines the function, the `return` statement (no parentheses are used here) tells what value the function should send back, regardless of whether this value is to be assigned to a variable or immediately used by the main program. Insert the function block near the top of your program, before you begin reading in the file. Then, within the `for` loop but after the line is split into separate fields, add these calls to your new function:

```
      LatDegrees = decimalat(ElementList[2])
      LonDegrees = decimalat(ElementList[3])
      print "Lat: %f, Lon: %f" % (LatDegrees,LonDegrees)
```

Notice how clean and compact the program becomes. Your function is called twice each time through the loop, once for longitude and once for latitude. You send it a string to operate on, and it provides the floating point value of the converted string, making this available for assignment to a variable.

Finally, adding `OutFile.write()` statements to your program, whether in place of or in addition to the `print` statements, will write this output to your `OutFile`. Here is the new version of the program, also available in the examples folder as `latlon_4.py`:

```
#!/usr/bin/env python
#
# This program reads in a file containing several columns of data,
# and returns a file with decimal converted value and selected data fields.
# The process is: Read in each line of the example file, split it into
# separate components, and write certain output to a separate file

import re # Load regular expression module, used by decimalat()

# Functions must be defined before they are used
def decimalat(DegString):
   # This function requires that the re module is loaded
   # Take a string in the format "34 56.78 N" and return decimal degrees
   SearchStr='(\d+) ([\d\.]+) (\w)'
   Result = re.search(SearchStr, DegString)

   # Get the captured character groups, as defined by the parentheses
   # in the regular expression, convert the numbers to floats, and
   # assign them to variables with meaningful names
   Degrees = float(Result.group(1))
   Minutes = float(Result.group(2))
   Compass = Result.group(3).upper() # make sure it is capital too

   # Calculate the decimal degrees
   DecimalDegree = Degrees + Minutes/60

   # If the compass direction indicates the coordinate is South or
   # West, make the sign of the coordinate negative

   if Compass == 'S' or Compass == 'W':
      DecimalDegree = -DecimalDegree
   return DecimalDegree
# End of the function decimalat() definition

# Set the input file name
InFileName = 'Marrus_claudanielis.txt'

# Derive the output file name from the input file name
OutFileName = 'dec_' + InFileName

# Give the option to write to a file or just print to screen
WriteOutFile = True
```



```
# Open the input file
InFile = open(InFileName, 'r')

HeaderLine = 'dive\tdepth\tlatitude\tlongitude\tdate\tcomment'
print HeaderLine

# Open the output file, if desired. Do this outside the loop
if WriteOutFile:
   # Open the output file
   OutFile = open(OutFileName, 'w')
   OutFile.write(HeaderLine + '\n')

# Initialize the counter used to keep track of line numbers
LineNumber = 0

# Loop over each line in the file
for Line in InFile:
   # Check the line number, don't consider if it is first line
   if LineNumber > 0:
      # Remove the line ending characters
      # print line  # uncomment for debugging
      Line=Line.strip('\n')

      # Split the line into a list of ElementList, using tab as a delimiter
      ElementList = Line.split('\t')

      # Returns a list in this format:
      # ['Tiburon 596', '19-Jul-03', '36 36.12 N', '122 22.48 W',
      # '1190', 'holotype']
      # print "ElementList:", ElementList  # uncomment for debugging

      Dive    = ElementList[0]
      Date    = ElementList[1]
      Depth   = ElementList[4]
      Comment = ElementList[5]

      LatDegrees = decimalat(ElementList[2])
      LonDegrees = decimalat(ElementList[3])
      # Create string to 5 decimal places, padded to 10 total characters
      # (using line continuation character \)
      OutString = "%s\t%4s\t%10.5f\t%10.5f\t%9s\t%s" % \
                  (Dive,Depth,LatDegrees,LonDegrees,Date,Comment)
      print OutString
      if WriteOutFile:
         OutFile.write(OutString + '\n') # remember the line feed!

   # another way to say LineNumber=LineNumber+1...
   LineNumber += 1 # this is outside the if, but inside the for loop

# Close the files
InFile.close()
if WriteOutFile:
   OutFile.close()
```

Scan through the program and make sure that you understand—or can figure out—what each section does. There are a few additional adjustments here and there in the example program which we haven't explicitly covered, but these should make sense in context. For instance, the line number counting is slightly different, taking advantage of a shorthand operator, `+=`, for incrementing a variable by a given value without writing the variable name twice. Many other programming languages also support this operator. Thus, `LineNumber += 1` does the same thing as `LineNumber = LineNumber + 1`, but with less typing. We will use this construct from here on.

Run the program at the command line, and use the `less` command to inspect the contents of the file it generates, `dec_Marrus_claudanielis.txt`.

If you have problems getting the program to execute properly, take this opportunity to practice your proofreading and debugging skills. Is the program running, but not doing what it should? Is your indentation consistent and correct? Is there punctuation missing—for example, colons after the `for` statements, the function `def`, and the `if` statements? If there are errors when you run it in the terminal window, Python will list the offending lines. Look in the vicinity of these lines for problems—the error is often located *above* the indicated line number. Also try commenting out portions of code to isolate the location of the problem, or printing out status variables to see whether the program enters the loop, what the list of elements looks like, and whether the regular expression is working. As always, if you get stuck or frustrated, consult Chapter 13 for debugging tips or seek help at `practicalcomputing.org`.

Here is a summary of the commands used when writing to a file:

```
OutFileName = "Outputfile.txt"
OutFile = open(OutFileName, 'w')
OutFile.write(DataLine + '\n')
OutFile.close()
```

Packaging data in a new format

Reformatting your file into a different tabular format, as you have done above, is a skill that you will use repeatedly. Hang on to the `latlon_4.py` script and modify it to suit your own purposes. Our ultimate goal in this chapter, though, isn't to write another tab-delimited file; rather, it is to write a KML file that can be understood by Google Earth. You are most of the way there, having read and converted the actual coordinate data. Now you need to embed these converted coordinates within the formatting characters of a KML file.

Examining markup language

The Google Earth file format is called Keyhole Markup Language, or KML. Keyhole is the name of the company that originally developed the format (before being acquired by Google), and markup language refers to a variety of formats for marking up or tagging information in a text file. Other markup languages include XML, a general data format, SVG, a graphics format, and HTML, the language of web sites. KML itself is actually just a variety of XML.

These markup languages have in common a style of bracketing or annotating each data element with a pair of opening and closing **tags**. These tags, which are themselves enclosed within angle brackets, indicate the nature of the enclosed data:

```
<element_name>Element Content</element_name>
```

For example, you might create a file for your samples that records species name, depth, and location in an XML format as follows:

```
<sample>
 <species>Marrus claudanielis</species>
 <depth>-518</depth>
 <location>-70.238, 39.940</location>
</sample>
```

In this example, a sample tag delimits the data for a sample. The sample data include species, depth and location records, with similar corresponding tags. Notice that each opening tag is matched by a closing tag, and that the closing tag is the same as the opening tag except that its contents start with a slash (/). You can think of the opening and closing tags as a pair of parentheses or brackets. If you open one, you need to close it later; also, nested tags need to be closed in the reverse order they were opened in (just as `[{}]` makes sense, but `[{]}` doesn't.)

You are now going to generate some XML output from the table of values that you have been working with. You might be surprised how easy it will be to modify your existing converter program so that it outputs KML instead of a table of plain text values.

VIEWING YOUR KML FILE To see your data plotted visually on a globe, you will need to install the free Google Earth application. If you don't have it already, you can get it from `http://earth.google.com`. Once you have the program running, you can open KML files and the points will appear on your map.

INTERNATIONAL CHARACTERS Not all words can be represented by the letters `A-Z` and ASCII standard symbols (see Chapter 1 and Appendix 6). If your program needs to access and display accented characters and special symbols (¢ ° £), you will need to add another special header line immediately below the shebang line, to tell it to expect Unicode (in this case the common variant UTF-8):

```
#!/usr/bin/env python
# coding: utf-8
```

In Python versions less than 3.0, strings that you want to be interpreted as Unicode should be preceded by a lowercase `u`, as in `u"Göteborg"`, in a manner similar to designating raw strings by preceding them with a lowercase `r`.

Preserving information during conversion

When you write your output file, not all of the fields from the original file will be required for the data to be read and displayed by Google Earth. However, it is a good idea to put all the data that is in the input file into the output file, if possible. If you put just what you think you need in the output file, chances are that at a future date you will want to use that file in a new way that you did not originally intend, which may require more data fields than you had decided to save. (On the other hand, sometimes it is not possible to preserve all the information from the input file, because of formatting restrictions on the output file; and at other times, the whole point of the file converter may be to skim off a small fraction of relevant data from a large and unwieldy input file.) Since KML files don't have fields corresponding to all the columns in your input file, you will work around this by putting each original line into the general description field of the KML file.

Converting to KML format

KML file format

There are three main sections in a KML file (and in XML files in general). These are a short header region, the repeated data entries, and a short footer region.

Header The header for a KML file can be just three lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
<Document>
```

The first line indicates how the contents of the file are formatted, indicating that it is an XML file. This line stands on its own, and is a special type of tag that does not need to be closed later. The second line is the opening tag for the `kml` portion of the code, and the third line is the opening tag for the `Document` content. These last two lines will have closing tags, `</Document>` and `</kml>`, in the footer at the very end of the document. All of the location data will be placed between the opening and closing `Document` tags.

Placemark The main section of the KML file will include the data points that are to be plotted on the map:

```
<Placemark>
   <name>Tiburon 596</name>
   <description>Tiburon 596  19-Jul-03  36 36.12 N  122 22.48 W  1190
         holotype</description>
   <Point>
      <altitudeMode>absolute</altitudeMode>
      <coordinates>-122.374667, 36.602000, -1190</coordinates>
   </Point>
</Placemark>
```


Each multi-line Placemark corresponds to one line from the original input text file. Between each open-close pair of `<Placemark>` tags are nested tags for the name, coordinates, and description. The new version of the program will create a Placemark—consisting of the opening and closing `<Placemark>` tags and everything between them—each time through your loop.

Footer The last section of the file is just two tags which close the `<kml>` and `<Document>` tags opened in the header:

```
</Document>
</kml>
```

Generating the KML text

You will now modify your `latlon` script to generate the header, Placemark, and footer strings. The modified file is saved in the examples folder as `latlon_5.py`. The header and footer will be generated and written to the file outside of the loop, since there is only one copy of each of these in the file. Within the loop, you will write the code to generate and write each Placemark string to the file.

Constructing strings with triple quotes rather than single quotes allows text to span multiple lines. The resulting string will include the line endings, so this is different than spanning multiple lines by escaping out line endings with \. Triple quotes are useful for defining blocks of text like the header and Placemarks.

The combined statement to generate the `PlaceMarkString` is:

```
PlaceMarkString = '''
<Placemark>
   <name>Marrus - %s</name>
   <description>%s</description>
   <Point>
      <altitudeMode>absolute</altitudeMode>
      <coordinates>%f, %f, -%s</coordinates>
   </Point>
</Placemark>''' % (Dive, Line, LonDegrees, LatDegrees, Depth)
```

The triple-quoted string that contains all the tags is combined with the string-formatting operator (`%`) to insert all five values into the Placemark at once. The insertion of the values in parentheses involves two strings (`%s`), two floating point numbers (`%f`), and a final string (`%s`). Because Google Earth will want a negative elevation for the depth, you must manually insert a dash before the `%s` that corresponds to the depth value. To use this entire string, you can either say `print PlaceMarkString` or `output.write(PlaceMarkString)`, depending on whether you are writing to the screen or to a file.

The entire program `latlon_5.py` is presented here, and is also available in the examples folder:

```
#!/usr/bin/env python
import re # Load regular expression module

# Read in each line of the example file, split it into
# separate components, and write output to a kml file
# that can be read by Google Earth

# Functions must be defined before they are used
def decimalat(DegString):
   # This function requires that the re module is loaded
   # Take a string in the format "34 56.78 N"
   # and return decimal degrees
   SearchStr='(\d+) ([\d\.]+) (\w)'
   Result = re.search(SearchStr, DegString)

   # Get the (captured) character groups from the search
   Degrees = float(Result.group(1))
   Minutes = float(Result.group(2))
   Compass = Result.group(3).upper() # make sure it is capital too

   # Calculate the decimal degrees
   DecimalDegree = Degrees + Minutes/60

   if Compass == 'S' or Compass == 'W':
      DecimalDegree = -DecimalDegree
   return DecimalDegree
# End of the function decimalat() definition

# Set the input file name
InFileName = 'Marrus_claudanielis.txt'

# Derive the output file name from the input file name
OutFileName = InFileName + ".kml"

# Give the option to write to a file or just print to screen
WriteOutFile = True
```



```
# Open the input file
InFile = open(InFileName, 'r')

# Open the header to the output file. Do this outside the loop
Headstring='''<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<kml xmlns=\"http://earth.google.com/kml/2.2\">
<Document>'''

if WriteOutFile:
   OutFile = open(OutFileName, 'w')    # Open the output file
   OutFile.write(Headstring)
else:
   print Headstring

# Initialize the counter used to keep track of line numbers
LineNumber = 0

# Loop over each line in the file
for Line in InFile:
   # Check the line number, process if you are past
   # the first line (number == 0)
   if LineNumber > 0:
      # Remove the line ending characters
      # print line  # uncomment for debugging
      Line=Line.strip('\n')

      # Split the line into ElementList, using tab as a delimiter
      ElementList = Line.split('\t')

      # Returns a list in this format:
# ['Tiburon 596', '19-Jul-03', '36 36.12 N',
# '122 22.48 W', '1190', 'holo']
    # print "ElementList:", ElementList  # uncomment for debugging

      Dive = ElementList[0]  # the whole string
      Date = ElementList[1]
      Depth= ElementList[4]  # A string, not a number
      Comment=ElementList[5]

      LatDegrees = decimalat(ElementList[2])  # using our special function
      LonDegrees = decimalat(ElementList[3])
# Indentation for triple-quoted strings does not have to
# follow normal python rules, although the variable name
# itself has to appear on the proper line
      PlacemarkString = '''
<Placemark>
   <name>Marrus - %s</name>
```

```
    <description>%s</description>
    <Point>
      <altitudeMode>absolute</altitudeMode>
      <coordinates>%f, %f, -%s</coordinates>
    </Point>
</Placemark>''' % (Dive, Line, LonDegrees, LatDegrees, Depth)

      # Write the PlacemarkString to the output file
      # This is indented to be within the for loop
      if WriteOutFile:
        OutFile.write(PlacemarkString)
      else:
        print PlacemarkString

   LineNumber += 1 # This is outside the if, but inside the for loop

# Close the files
InFile.close()
if WriteOutFile:
   print "Saved",LineNumber,"records from",InFileName, "as", \
      OutFileName   # Shown on the screen, not in the file
   # After all the records have been printed,
   # write the closing tags for the kml file
   OutFile.write('\n</Document>\n</kml>\n')
   OutFile.close()
else:
   print '\n</Document>\n</kml>\n'
```

Run the script, and open the resulting KML file in Google Earth.

This script will be another important tool in your toolkit. You should be able to modify this template to read nearly any column-organized data file and output it in another format. When doing this, it is usually a good idea to open the template file, do a Save As... with the name of your new project, and *then* begin your editing. That way you can make changes without worrying about breaking your working script.

SUMMARY

You have learned how to:

- Approach file parsing
- Read files with `open(FileName,'r')` and a `for` loop
- Remove trailing characters with `.strip('\n')`
- Parse strings into lists with `.split()`
- Access list elements with `[]`

- Write to files with `open(FileName,'w')` and `OutFile.write()`
- Continue Python lines with `\`
- Use regular expression searches with `re.search()` and `re.sub()`
- Access regular expression search results with `.group()`
- Create custom functions with `def function_name:`
- Generate XML and KML files
- Use triple-quoted strings (`"""  """`) to span multiple lines

Moving forward

- Find other types of input files containing latitude/longitude data and modify your program to support them. Make sure you understand how to allow for different formats of degrees and minutes (the `re.search()` function) and different numbers of header lines, to skip lines with comments marked by #, and to account for the possibility of footer lines at the bottom, all via the `if` statement within the `for` loop.
- Look at the HTML source of a web page presenting a table of values, and determine how you would replicate this by printing header, content, and footer sections.