Chapter 14

SELECTING AND COMBINING TOOLS

You have gained experience with a range of tools in the preceding chapters, but that is not all that is needed before you can apply them to new problems. You also have to be able to pick the right tool for each job. In some respects this is the more difficult skill to acquire. Here you will step back and consider how to decide which tool to use for which problems. This bird's-eye perspective provides an opportunity to review some of the skills covered earlier before moving on to other specialized topics.

Your toolkit

In the preceding chapters, you have become familiar with a variety of flexible and powerful tools for handling data. These tools fall into four broad categories:

- Regular expressions, to search and replace
- Shell commands, to interact with your computer at the command line
- Shell scripts, to combine and automate command-line operations
- Python programs, for more advanced processing

Given this range of options, there is almost always more than one possible approach to a given computing problem. Many tasks that could be addressed with a shell script, for example, could also be accomplished with a Python program. Sometimes a challenge is best addressed with a series of tools integrated into a combined workflow, rather than by one tool exclusively. While it is relatively easy to follow along with an example that walks you through how to solve a given problem with a given tool, it can be confusing to decide which tools to use when you approach a problem in the first place. Mapping out your analysis strategy and making these first important decisions is the focus of the present chapter.

We will present this information along with three decision charts. The charts begin with a general problem or task and proceed to possible solutions. Find the chart that most closely applies to your task, follow the path that matches your requirements, and then consider the indicated approach. These brief charts list methods drawn from previous chapters, and will also direct you to portions of later chapters that might be useful.

Categories of data processing tasks

Getting digital data

Text files are the universal currency of data analysis, so many of your tasks will involve gathering your data into a simple text format. This might involve getting input from the user (usually yourself), exporting data from another program, or extracting data from an online source.

Data from user input User input at the command line is a convenient way to accept input when the data are short and easily typed or pasted. The `dnacalc.py` program is an example where the user enters a value that is fed to the subsequent calculation. Other situations where you might consider user input are a script to reverse-complement a DNA sequence, convert among oxygen-saturation units, or change decimal latitude/longitude values to degrees, minutes, and seconds. User input can also tell the program what set of files you want it to act upon, as with the `sys.argv[]` variable used in your program `filestoXYYY.py`. In Chapter 16 you will see how to accomplish the same thing in the `bash` shell, using `$1` to represent user input within your shell scripts.

Data from the Internet How you interact with Internet resources depends on your needs—in particular, the number of files you anticipate processing and how often you will need to access them (Figure 14.1). If you anticipate needing to grab only a few files and process them into a usable format just once, then the simplest approach is to call up the Web page, view the source, and copy and paste the source into a new text document. From there you can use a series of regular expressions to reformat the data. The process of accessing Web sources is described in Chapter 9, and regular expressions are described in Chapters 2 and 3 and summarized in Appendix 2.

If you need to extract many files from the Web—whether a data series, some number of images, or a set of Web pages—it will probably be best to automatically download these files with `curl` (or alternatively with `lynx`, or in Linux, `wget`) in a shell window. This is much more convenient than clicking through the source code of a large number of pages in a web browser. The `curl` command is described at the end of Chapter 5. You can either use `curl`'s ability to gather many files at once, or you can save many `curl` commands together in a shell script, as shown in Chapter 6.

If you are regularly retrieving data from the Web—for example, grabbing a daily record of temperatures—then it may be worth writing a Python script using

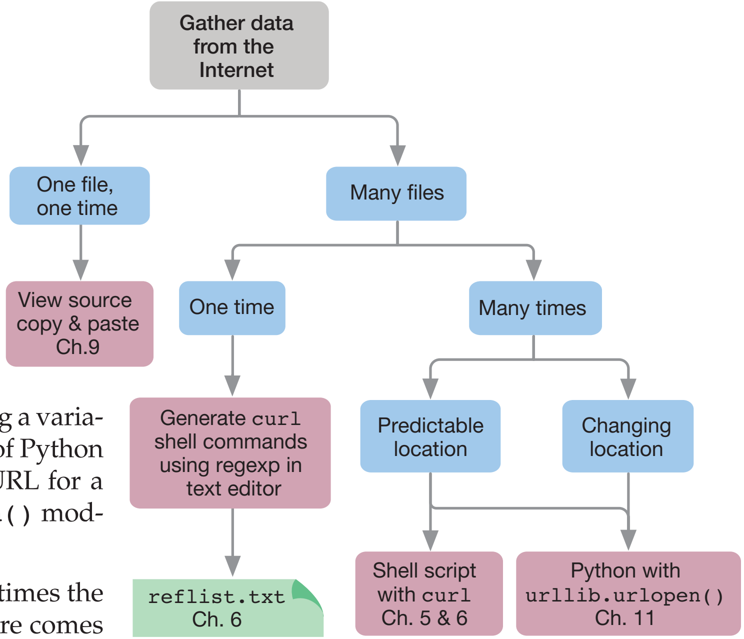


FIGURE 14.1 Decision chart for gathering data from the Internet Gray boxes indicate the general problem. Blue boxes are progressively more specific descriptions of the task. Maroon boxes are appropriate tools. Green boxes are some specific example scripts that demonstrate the solution.

`urllib.urlopen()`. This approach can also accommodate more complex Web addresses that cannot be generated by `curl`. For instance, many URLs for time series data are built using a variation of the date, so you can write a bit of Python code that automatically generates a URL for a particular day. The `urllib.urlopen()` module is briefly illustrated in Chapter 12.

Data from other programs Sometimes the stream of data that you wish to capture comes as the output of another program. Working at the command line, you can either capture output to a file using the redirection operator (> or >>), or else send the output to another shell command using the pipe operator (|). For example:

```
history | grep "pcfb"
```

Shell operations like these can sift through a long stream of output for items of interest. For more examples of combining shell commands using pipes and functions, see Chapter 16 or consult Appendix 3.

Redirects and pipes don't work in every case to connect command-line programs, however. For example, if the first program you wish to use outputs data to the screen, but the second program requires the name of a data file as input, a pipe isn't suitable for combining them into a single command. The second program would interpret the output of the first program as a filename rather than as the data itself.

Data directly from hardware Many instruments and sensors interface with a computer through a serial port. The data stream from this equipment is often recorded with special software that is supplied by the equipment manufacturer. From there, it can usually be exported in plain text format. Often the software provided with such equipment is quite limited and can't be combined with other tools into automated workflows. When you want to use the instrument in a way that the supplied software won't allow, it may be possible to intercept the data stream directly from the serial port and then parse it any way you like. Chapter 22 gives some background on how to tap directly into these devices and interact with the physical world with custom-built electronics.

Data from spreadsheets Many people think of spreadsheet software as fundamental to managing datasets. In fact, spreadsheets are more suited to small one-off projects and simple record-keeping than they are to large analyses and complex datasets. They don't make a good central data repository because so few programs can open and manipulate the complex file formats they use. In addition they are inefficient and sometimes incapable of handling large datasets.

The first step to handling spreadsheet data in the context of a larger analysis is usually to resave the data in a basic text format. The brute force method for resaving the data is to open that file in its native program (e.g., Excel, OpenOffice, Numbers) and export the data as a text file delimited by either tabs or by commas (that is, a CSV file). While this works fine for one or two files, repeatedly clicking through the menus of a spreadsheet program can be quite tedious for even a small number of files. If you have MATLAB installed, you can write a file conversion script with MATLAB's `xlsread` command to convert Excel files to raw text. Another option is to install the `xlrd` Python package to access spreadsheet file contents. (General instructions for software installation are in Chapter 21, but briefly, you can download the source code from `http://pypi.python.org/pypi/xlrd`, uncompress the archive by double-clicking, `cd` into that directory, and type `sudo python setup.py install`.) Other spreadsheet conversion packages include the Perl-based `xls2csv` function and the PHP-based `phpexcelreader`. Alternatively, you could write a macro for OpenOffice or Excel to read in all your files and export them back out.[1]

Newer spreadsheet files with the `.xlsx` file extension are compressed archives containing XML files, which you encountered in Chapter 10. Buried among the documents in that archive is a text file containing your data. If the Python `xlrd` function does not work for your files, it might be possible to open these XML files in your text editor or with a Python script and extract the data of interest. From the command line, try uncompressing an `.xlsx` file and looking in the `xl/worksheet` directory:

```
host:~ lucy$ unzip SpreadsheetDataA.xlsx
Archive:  SpreadsheetDataA.xlsx
  inflating: [Content_Types].xml
  inflating: _rels/.rels
  inflating: xl/_rels/workbook.xml.rels
  inflating: xl/workbook.xml
...etc...
host:~ lucy$ cd xl/worksheets
host:worksheets lucy$ ls
sheet1.xml  sheet3.xml  sheet5.xml  sheet7.xml  sheet9.xml
sheet2.xml  sheet4.xml  sheet6.xml  sheet8.xml
```

[1] A macro is a script built within another program's own internal scripting system. OpenOffice, ImageJ, AppleScript, and Automator can create macros.

It will probably look like a mess, but `sheet1.xml` will contain the first sheet of data in your file. A shell script could fairly easily perform this unzip command on all your files, and copy the file `xl/worksheets/sheet1.xml` to `sheetA.xml`. There is a built-in Python module called `xml` which is not covered here but which gives some capability to work with XML files.

Of course, an easy way to avoid dealing with these conversions is to avoid spreadsheet data files throughout your data analysis.

Reformatting text files

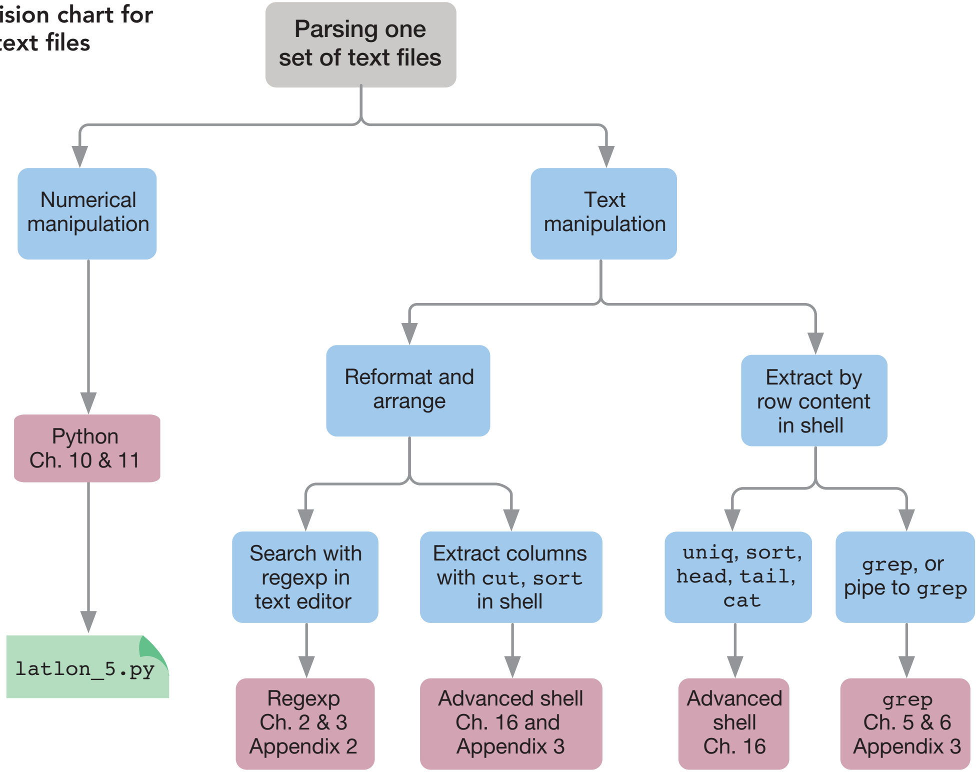
The most common starting point for data processing is a text file in the "wrong" format—the data are there, but they aren't arranged in the right way for the intended use. You can use any number of general-purpose tools to reformat data in text files, and you don't need specialized software to open a text file and examine it. Whenever possible, you should save your data and request data in a plain text format instead of other specialized formats.

There are so many options for reformatting data files that it is impossible to capture all the possibilities in a single chart. We have summarized some of the primary approaches in Figures 14.2 and 14.3. The first thing to consider is whether you will just be doing a given reformatting job once on a few files, in which case regular expressions in a text editor will suffice, or whether you expect to perform the reformatting on many files or on many occasions, in which case a script is the way to go. Keep in mind that you often need to go back and reanalyze a data set—perhaps some values need to be calibrated, or you must add some important last-minute measurements that weren't initially available. Even if it will take you twice as long to create a script as to perform the same operation manually in a text editor, the script is often a worthwhile investment of time. In addition to enabling quick reanalyses, the script itself serves as documentation of how the data were modified, which can be critical for writing up your results.

Starting with regular expressions If you have only one file to process, you can use regular expressions (also known as regexp) in a text editor to rearrange and extract data elements, as explained in Chapters 2 and 3. When considering your search and replace operation, a single search expression is often not the easiest way to do a file conversion—so don't think that you have to solve it all in one brilliant regexp. You can use several expressions in a row to take care of different aspects of the reformatting. You should document the changes you make by keeping a record of the replacement patterns in your notebook and in a notebook file. For example, if you perform several replacements in succession, you can copy the search and replace terms into a separate document. This will be useful if you have to repeat the transformation or if you end up writing a Python script to recapitulate those replacements automatically. In `TextWrangler` you can also save searches for later, or pick from a list of recent search and replace combinations.

If you have already gone through the introduction to regular expressions in Chapters 2 and 3, then the reference table in Appendix 2 is a good way to remind yourself of the syntax and strategies.

FIGURE 14.2 Decision chart for parsing one set of text files



Command-line operations Another way to work interactively with your files is through the command line. This is useful when several files are involved—for example, when joining files with the `cat` command or extracting certain lines with `grep`, as described in Chapters 5 and 6. Working with text at the command line is also an important skill to have if you are manipulating files on a remote computer, as described in Chapter 20.

Some advanced shell functions can be surprisingly powerful for text manipulation, in particular those described in Chapter 16. These include the `cut` command to extract certain columns of data, the `sort` command, which lets you arrange lines either alphabetically or numerically, and the `uniq` command, which reduces a data stream to a list of non-repeated entries, and which can also count the number of entries in the process. These commands can be chained together using the pipe operator (|), introduced in Chapter 5 and elaborated upon in Chapter 16. Shell commands can be gathered together into a text file to serve as a script, as described in Chapter 6. Use the `history` command to review your recent shell operations so you can edit them together into a script file. Shell operations can also be combined into frequently used aliases (shortcut commands) or functions (multiline shortcuts, including the possibility for user input). These approaches

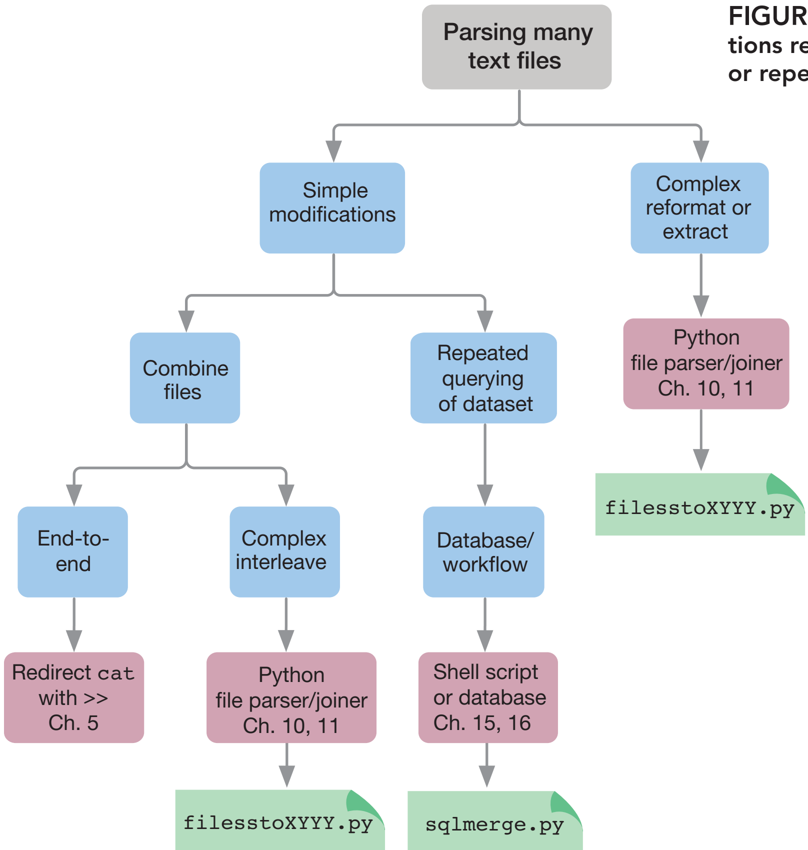


FIGURE 14.3 Decision chart for operations requiring processing of many files, or repeated processing of a few files

are most useful for relatively simple extractions on large numbers of similar files. Renaming or moving files is also an ideal role for a shell script or function.

You might think that graphics files need to be handled through a graphical user interface. However, shell scripts can readily operate on graphics files, using many of the command-line tools presented in Chapter 19. If your workflow involves images, be sure to investigate the capabilities of `sips`, ImageMagick (via `convert` and `mogrify`), and `exiftool`.

Python scripts

Python scripts are powerful tools for implementing and automating text manipulations like parsing and formatting, as well as for more sophisticated analyses. Many modules are available for specialized operations, and Python can even be used to control other programs. The example programs and scripts from this book, summarized in Table 14.1, are intended to serve as templates from which you can develop programs suited to your own research. They include three categories of interaction: user input, reading single files, and reading multiple files.

For interacting with user input in a Python script, either to create an advanced "calculator" or to operate on a set of user-specified files, you can use the `raw_input()` function or else the `sys.argv[]` variable described earlier. For more involved operations, you will typically be reading from and writing to data files using the `open` command:

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

This will typically be followed by a `for` loop to cycle through the lines in the file:

```
for Line in Infile:
    # process each line here
```

Some examples of file reading and writing in Python are presented in `latlon_5.py`, `mylatlon_4.py`, and in `filestoXYYY.py`.

For many operations, a Python program with a single `for` loop is sufficient. However, for more in-depth calculations or for synthesizing data across files, you will probably need to use one loop to read the file into a list or dictionary, before generating output based on a composite of the values. A second `for` loop can cycle through to print the output lines or to save them to a file as explained in Chapter 10.

To combine shell commands with Python scripts, you can either include a program name in a `bash` script or call shell commands using Python's `os.popen()` functions, as in the `exifparse.py` example script mentioned in Chapter 16.

General considerations

One of the biggest pitfalls encountered as people take on more complicated analyses is to not keep track of what they have done. A scientific result is little more than a rumor if you can't tell someone how you got it, and you don't want to waste time second-guessing what you did if you need to do it again. As you work through analyses, keep a careful record of your scripts, searches, and discoveries. Annotate this information as much as possible. Keep these records centralized. The simplest way to do this is to keep a single plain text notebook file for analyses, just as you would keep a written lab or field notebook. Paste in your commands and results, and explain why you did things the way you did.

The best program is often the one you are most familiar with; by scavenging and repurposing from scripts you understand—including those presented in this book and other scripts you find online—you will build up a set of tools that should serve you well for a variety of tasks. A `HandyShellCommands.txt` file is a good place to jot down useful commands. You can quickly `grep` through this file to remind yourself of relevant commands for various operations. Without regular use, your shell proficiency and scripting skills can get rusty, so a combination of good note-taking and what amounts to regular exercise can keep you in good shape.

TABLE 14.1 Example programs and scripts discussed

Shell examples	
`shellfunctions.sh`	A list of shell functions to use as starting points for adding tools to your own `.bash_profile`
`shellscripts.sh`	Examples of long shell commands discussed in the text, including use of the `curl` function and editing your `PATH`
Python examples	
`asciihexbin.py`	Print a table showing ASCII or Unicode, hexadecimal, and binary values
`bootstrap.py`	Use the `random` module to create bootstrapped datasets (resampled from original data with replacement); could be converted to a module for use in other programs
`compositioncalc2.py`	Loop through the unique elements of a string (e.g., protein or DNA sequence) and calculate the percent composition of each element
`dnacalc2.py`	As above, but calculate additional properties of the sequence using different formulas depending on the sequence length
`exifparse.py`	Use Python to run shell commands, including the `exiftool` program; extract quantitative metadata from a series of images and print in tabular format
`filestoXYYY.py`	Take a series of files as input, extract columns of data, and join the columns together in one file
`latlon_5.py`	Convert a delimited file of locations to a Google Earth KML file
`matplotCTD.py`	Within Python, use the `matplotlib` module to plot data from a file
`mylatlon_4.py`	Read in a delimited file, convert format, and insert entries into a `mysql` database, using the `MySQLdb` module
`proteincalc.py`	For each character in a string, look up a corresponding numeric value from a dictionary and add that value to a sum
`seqread.py`	Load data from a file into a list and dictionary, using the first character of the line to determine which values become keys
`serialtest.py`	Demonstration of reading from a serial port using Python's `pyserial` module
`sqlmerge.py`	Extract values from one table in a `MySql` database, and use them to extract values from another table; merge the values into a delimited print-out

SUMMARY

You have:

- Reviewed how to gather data from users, Web sites, and other programs
- Learned some of the options for extracting data from a spreadsheet
- Reviewed the options for reformatting text files

Moving forward

- Create a text file containing some of your favorite commands and update it when you discover something new and useful.
- Make copies of the programs and scripts that are most relevant to your research, rename them, add heavy annotations, and adapt them to suit your purposes.

Chapter 15

RELATIONAL DATABASES

There has been a strong emphasis throughout this book on storing data in easy-to-understand and portable plain text files. In general, plain text files are well-suited for many scientific needs. There are other options, though, that have many of the same advantages of text (open standards, readability, and wide support), but which add capabilities for extracting and synthesizing information from large or disparate data. This chapter illustrates these relational database management systems, with a focus on MySQL. Before introducing databases, we present some general considerations on deciding how to store data.

Spreadsheets and data organization

Developing an effective strategy for the storage and organization of data and data files is a critical part of nearly every scientific endeavor. Many data can be stored efficiently in two-dimensional grids, and we start here with some general advice on these 2-D data files. These grids usually have columns with different types of measurements, and rows with different samples or observations. Character-delimited text files represent two-dimensional data by placing each row of data on a line and separating data from different columns by a delimiter, usually either a tab or a comma (in the case of `.csv` or "comma-separated value" files). Although we have focused on text files for much of this book, we do realize that for many people, a spreadsheet is the primary way they enter, interact with, and analyze their results. A spreadsheet is just a graphical representation of this two-dimensional grid, with tools for editing and calculating values.

There is a tendency to organize character-delimited text files and spreadsheets the way you would arrange the table of a publication, with separate headers and sub-tables for different experiments (Figure 15.1A), or with alternating rows of treatment and control (Figure 15.1C). This patchwork approach to grids is rarely suitable for subsequent work like analyzing the data, importing them into other programs, or performing numerical or statistical analyses. Problems usually arise when information for a particular observation is spread across multiple rows. In contrast, the most general and flexible way to store data in tables is to make sure that each row has all the data needed to interpret that row.

In suboptimal approaches of the type shown in Figure 15.1A, the row shown in green contains descriptors of the data in rows that follow. In essence, the data are separated into multiple tables stacked on top of each other, and interpreting a row with data requires knowing something about the descriptor row somewhere above it. A preferable approach, shown in Figure 15.1B, would be to give the descriptors their own column, and repeat the descriptor within each row to which it applies. This way all the information about a row is found right within a row, and a program can read the data directly without parsing descriptors separately from other pertinent rows.

Another suboptimal approach, shown in Figure 15.1C, is to put the pairs of control and treatment values (or background and signal) on alternating lines, as represented by the blue and green boxes. If possible, these values should be put in the same row, as in Figure 15.1D. These modifications turn the file into one large grid with a single header.

The reasons for organizing character-delimited text files and spreadsheets as one large grid are numerous. Nearly all databases, statistics programs, and analysis programs such as MATLAB and R import and organize data in this format. Even in a spreadsheet, one formula added into a new column can draw information from other values located relative to it in the same row, instead of from miscellaneous places in the table. This allows you to use a single formula down the entire length of a column, making your analyses and graphing operations as efficient as possible. If you were to draw arrows to the cells that

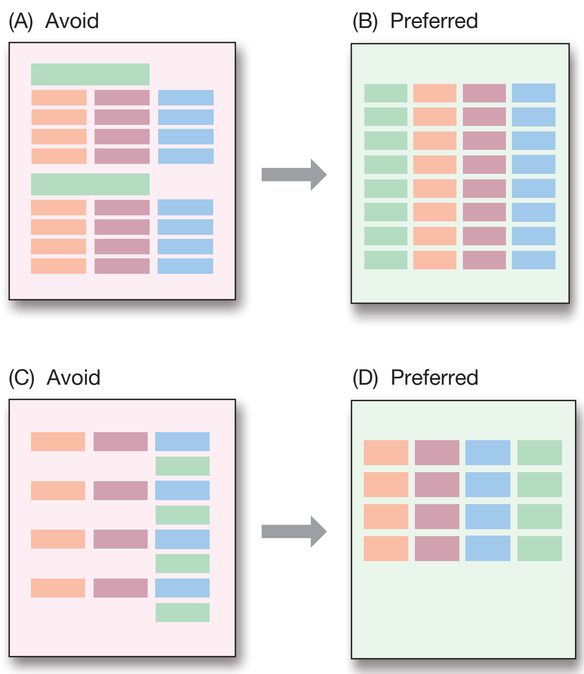


FIGURE 15.1 Approaches to organizing data in spreadsheets and character-delimited text files Each colored block represents a different type of data or a different comparable group of measurements such as date, temperature, category of treatment, or control and response variables. (A) and (C) show common but difficult-to-analyze approaches to organizing data. (B) and (D) are possible ways to reorganize the data so that each column contains a single type of information, and each row contains all the information relevant to a record.

are being used to calculate your results, would you get a simple network, or something that looks like a subway map? *In general, all data in a column should hold the same type of values, and each row should hold information that corresponds to a particular measurement.* The type of measurement, rather than being indicated by its position in the table, can be designated by a separate column devoted to that purpose.

When you are using a spreadsheet program, there are a few things you can do to improve clarity and facilitate the analysis of the data. Raw values should be entered into the cells; don't do any calculations or conversions before entering the data. Let the spreadsheet do the work of making conversions, and keep your data as unmodified as possible. This will save time and also provide a better record of the processing that the data have undergone during analysis. Likewise, don't type values directly into a formula; keep the values in their own cells and refer to those cell locations with your formulas. Again, this will make it simpler to update data and calculations, and easier to keep track of processing steps.

Some data cannot readily be coerced into a two-dimensional table with uniform columns, or at least not without duplicating excessive amounts of information within the file. Data with a nested structure, such as phylogenies, are not well suited to tabular organization. In other cases, it may be necessary to repeat most information from row to row, rather than just one or two columns. If you find yourself in such a situation, where a simple two-dimensional grid is insufficient or inefficient for your needs, you should think about moving beyond spreadsheets and character-delimited text files, and instead investigate keeping your data in a relational database, as we will discuss in the remainder of this chapter. (Of course, grids and databases are not the only way to store data in text files; many general file formats have been developed for complex datasets. The most widely used of these formats is XML, which was introduced briefly in Chapter 10 but isn't covered further in this book.)

Data management systems

The pervasiveness of spreadsheets can give the false expectation that all data are best stored in a single two-dimensional grid. This is not the case. For example, imagine that you want to track information about field sites and about multiple specimens observed at each field site. If there are many specimens per field site, shoehorning all the data into a single grid would require repeating the site information many times (once for each specimen found at a site). If you want to update the field site information, you will need to change it in many places. This approach has other problems besides redundancy. If you have complex data for several different but related elements of a study, these different data may not correspond in such a way that each record can be placed into the row of a single table. If you create individual files or grids within files to store different types of associated data (such as specimen data that include geographic coordinates for collection sites, molecular sequences for multiple genes, and one or more photographs), you cannot easily retrieve information from one of these tables based on data stored in

This chapter stands on its own, so if relational databases are not applicable to your analysis needs at this time, you can skip ahead without affecting your ability to use other sections of the book.

another table. Nonetheless this is often exactly what you would want to do.

This is where relational databases come in. A relational database management system, or RDBMS, is a server program that runs continuously in the background and manages one or more databases. These databases are collections of structured information. Although databases are stored as files, the user doesn't interact with the files directly—the management system acts as a middleman. It takes care of the creation, organization, and optimization of the files, as well as all direct interaction with them. It also listens for requests to add, edit, or look up data. These requests can come from other software on the same computer, or the computer can be configured to accept requests over network connections, so that the database and program using it don't even need to be in the same location. Commercial database management systems include FileMaker, Microsoft Office Access, Microsoft SQL Server, and the Oracle software suites. Open source options include MySQL (maintained by Oracle), PostgreSQL, and SQLite. These management systems are designed to work with a wide range of database sizes, from dozens of entries to billions. Complex tricks are used behind the scenes to process requests quickly and to optimize file organization for speed and memory efficiency. It is generally much faster to find a particular piece of information in a database than it would be to scan through a large text file. The files used to store the data are different from system to system, but because the user never directly interacts with these files, the differences don't usually matter.

Interactions with all modern database management systems are performed in a database language known as **Structured Query Language**, or SQL.[1] This language includes commands, functions and variables, and it follows a formal syntax. Since nearly all systems use a closely related variation of SQL, if you learn the basics of SQL once, you'll be well prepared to use most database software. Database management systems come with command-line and graphical interfaces for creating and interacting with databases by submitting SQL commands, but you can communicate with them in other ways as well. In addition to their direct interfaces, such systems also have back-door interfaces that allow for interaction with their databases from within other software packages. R, MATLAB, Python, web servers, and many other tools can be configured to interact directly with a database. This obviates the need to import and export data to and from files. In Python, for example, the ability to interact with a database is added with modules, one of which will be introduced later in this chapter. SQL queries can then be constructed as strings and sent to the database management system. Any data that are returned can be accessed from within Python.

In addition to accessing data, there are several important logistical advantages to using a database management system. Database files are centralized, so it is easy to back them up, and this avoids nightmare situations where there are redundant

[1] The proper pronunciation of SQL is open to debate, but here we pronounce each letter.

versions of data files and different ones are updated independently. The RDBMS takes care of a lot of the overhead of database management that you would otherwise need to build into your software. It also makes a good centralized warehouse. If you are working a project that involves several analysis programs, getting them to all talk with each other through multiple data file intermediates can be one of the biggest challenges. If they can all talk via the central database server, then it is much easier to pass data between them. More than one program can even access the database at the same time.

Relational databases don't only change the way that information is stored and retrieved on the computer, they also allow for important flexibility in the way that the data are organized and how you interact with them. The driving concept is that each piece of information is stored only once, and then linked through relations to other pieces of data rather than copied. This makes updates easier, reduces the chance of inconsistencies, and makes the database more efficient in term of both memory and computation. Most of these advantages become more apparent for larger and complex analysis projects. For smaller projects, database systems may be overkill.

Anatomy of a database

One RDBMS server can host any number of databases (Figure 15.2). Each database in turn contains two-dimensional tables that hold the actual data. Each column in a table has a different type of data, and each row contains a record. In this respect, database tables are recognizable to anyone who has used a spreadsheet—although as you will see later, databases provide much more powerful tools for interacting with the data and linking data across tables. There can be one table or hundreds of tables in a database, each containing a collection of records that pertain to a particular category of information. A database for a library might, for instance, have one table for books, one table for patrons, and another table describing which patrons have checked out which books.

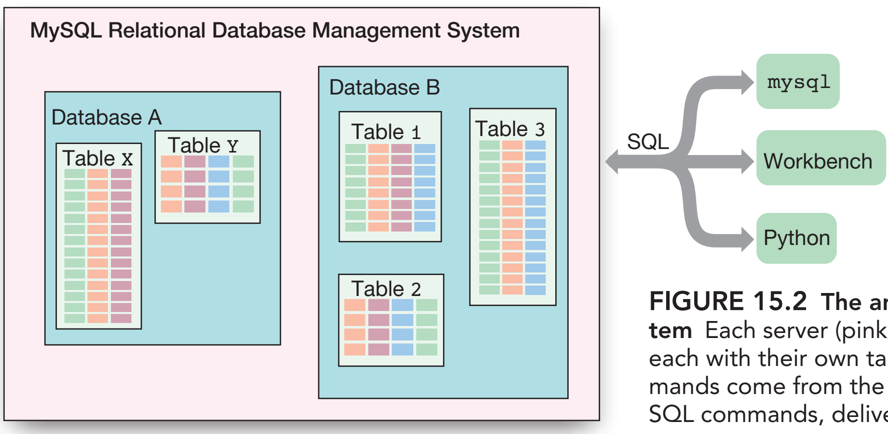


FIGURE 15.2 The anatomy of a database system Each server (pink box) can store databases, each with their own tables. Queries and commands come from the outside world by means of SQL commands, delivered by a variety of means.

TABLE 15.1 Common RDBMS data types

Data type	Description
`INTEGER`	An integer ranging in value from -2147483648 to 2147483647; `INT` can be used as an abbreviation for `INTEGER`
`FLOAT`	A floating point number, including scientific notation: `3.14159` or `6.022e+23`
`DATE`	A date in `'YYYY-MM-DD'` format
`DATETIME`	A date and time in `'YYYY-MM-DD HH:MM:SS'` format
`TEXT`	A string containing up to 65535 characters
`TINYTEXT`	A string containing up to 255 characters
`BLOB`	A piece of information encoded in binary, including images or other non-text data; there are four sizes of blob data types, with different storage capacities

The type of data in each column of a database table must be specified, and an error will result if you try to add data that don't conform to the specified type. Many of the data types used in databases are the same as those you have already encountered in Python (see Chapter 7), such as integers, floats, and strings. The naming of the types is a little bit different, and there are some types that are available in one context but not another. The most frequently used data types are listed in Table 15.1; additional types can be found at `dev.mysql.com/doc/refman/5.1/en/data-types.html`.

When you create a new table, one of the columns must be specified as the **primary key**, and each row of the table must have a unique primary key value which distinguishes that record or row. Using the primary key, you can unambiguously identify or extract any particular row of the table, in the same way that the key of a Python dictionary is uniquely associated with a particular value. The primary key is usually an integer value that is automatically computed by the database management system and stored when a new row is added.

Installing MySQL

Several excellent open-source relational database management systems are available, each optimized for different types of uses, but all perfectly adequate for many scientific tasks. Here we will provide specifics for getting started with MySQL, which is freely available and widely used in biology and science in general.

If you are using OS X or Windows, it is easiest to download and install MySQL directly from the project download page at `www.mysql.com/downloads`. There are different versions of the MySQL Server (the database management system itself) and a variety of ancillary files. To get started on your own computer, the two downloads you will need are MySQL Community Server, which is the actual RD-

BMS, and the MySQL Workbench, a graphical interface that facilities database maintenance and data visualization (Figure 15.3). If you have a computer running Ubuntu Linux, you can install the server through the Synaptic Package Manager, but you will still need to install the Workbench via the MySQL Web site.

There are a number of options available for the OS X download, including source and binary files built for different versions of OS X and for different computer hardware. The DMG archive is the simplest package format to install. Make sure that you get the correct file for your operating system version (e.g., OS X 10.6, 10.5, or 10.4; you can check the version of your operating system by clicking on About This Mac in the Apple menu). You have the option of choosing between 32-bit and 64-bit versions of the software. All Apple computers built since 2007 are 64 bit. If you have an older computer and aren't sure if it is 32 or 64 bit, the 32-bit version should work fine for most needs.

The OS X MySQL Community Server installation requires installing several different components, as described in the `ReadMe.txt` file provided with the installers. Run the two package installers; mysql-xxx.pkg installs the RMDBS, while MySQLStartupItem.pkg installs the launcher that starts the database software when the computer boots up. Drag the MySQL.prefPane to your /Library/PreferencePanes folder. Be sure to use the Library folder at the root level of your computer (this will require password authentication), not the folder of the same name in your home directory. This panel will allow you to start and stop MySQL within the System Preferences GUI.

There are a variety of command-line programs that come with MySQL. The installer will place these in `/usr/local/mysql/bin`, which is not in your default `PATH`. Edit your `.bash_profile` to add this directory to your path. (See Chapter 6 if you are rusty on how to do this. Briefly, you will add `:/usr/local/mysql/bin` to the end of the existing `export PATH` command.)

Consult the `ReadMe` file for your configuration information.

Restart your computer, open System Preferences, and click on the MySQL icon. In the MySQL preference pane it should say The MySQL Server Instance is running. If it is not running, click the Start MySQL Server button. If this results in an error window, consult the online MySQL documentation to troubleshoot the problem. If it wasn't initially running but works fine when you start it manually, make sure that you installed the MySQLStartupItem.pkg, that the Automatically Start MySQL Server on Startup box is checked, and then restart the computer again.

Once the MySQL server is running, install the MySQL Workbench. The installation is simpler than that for the server software: the program just needs to be dragged to the Applications folder. Launch the Workbench. The most important Workbench tools for getting started are the SQL Development tools on the left side of the window. It also has many advanced features that won't immediately be of use. Before you can interact with the database server you installed, you have to connect to it. Remember that database management systems don't automatically assume that a database lives on the same computer where it is being used, since it may also be accessed on a remote computer over the network. In fact, connect-

OTHER OPTIONS FOR SQL GUIs Another free GUI for interacting with SQL database systems is the Java-based, cross-platform SQuirrel SQL Client. You can download it for free from the links at `squirrelsql.org`. Be sure to install the Data Import and MySQL plug-ins as part of the installation process. If you choose this route, follow along with SQuirrel SQL wherever the Workbench is used. An inexpensive commercial option is Navicat, which also runs on nearly all platforms, but which has different program files for different databases.

ing to the local database is done as if the database server were on a different computer, but by using the special `localhost` address `127.0.0.1`, the connection is redirected back to the local computer. More detail on network connections and addresses, including `localhost`, is provided in Chapter 20.

Click on the New Connection icon in the SQL Development portion of the MySQL Workbench. All of the default values are for connecting to the local database. All you need to do is give the connection a name. Type `localhost` into the Connection Name box. Click Test Connection to make sure all the settings and installations are set up correctly. You will be asked for a password, which is blank by default. Click OK again. You should get a window that says Connection to MySQL at 127.0.0.1:3306 with user root, Connection parameters are correct. Click OK twice more, and you will get back to the main Workbench screen and will see the `localhost` connection you just created in the connections box.

Getting started with MySQL and SQL

There are many ways to connect to and interact with the MySQL server. In the previous section, you connected using the supplied MySQL Workbench graphical application. In this chapter we will use this graphical interface mostly to observe the changes to the database that are made through other types of connections. In many respects, the most convenient interface to the database is—yes, yet again—the command line. Later in the chapter you will also learn to connect to the database from within a Python program. The commands discussed in this chapter are summarized in Appendix 7 for quick reference.

Connecting to the MySQL server at the command line

Open a Terminal window, and enter the following commands (shown in bold):

```
lucy$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1426
Server version: 5.1.48 MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input.
mysql> SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
3 rows in set (0.00 sec)
mysql> EXIT;
Bye
lucy$
```

This launches the command-line program `mysql`, which provides an interface for interacting with the database server. It is not the server itself, as the name might imply, but, like the `MySQL Workbench`, a stand-alone program that can issue commands to the server and present the results. The `-u root` argument specifies that you want to connect to the database as the MySQL user `root`.[2] No password is needed since one hasn't been configured yet. MySQL users are different than the system users who have accounts on your operating systems, and they are configured completely separately. No network address is specified, so `mysql` assumes that you want to connect to the local MySQL server. Like the `Workbench`, `mysql` can connect to servers on other computers, but we won't get into the specifics of how to do that here.

If the connection is successful, you will get an introduction that describes a bit about the server, your connection to the server, and some other status information. If the connection isn't successful or the program won't run, make sure that the MySQL server is running and that `mysql` is in your `PATH`. You will then get a `mysql>` prompt. Like the shell prompt that takes `bash` commands or the interactive python prompt that takes Python commands, the `mysql>` prompt is waiting for database commands in SQL. As was the case with the other languages we have covered, SQL has far more features than we'll be able to address here. The goal is simply to provide you with enough information to get started, implement some simple projects, and help you determine whether a relational database is something that would be of use for your work.

The first thing to notice about SQL is that commands end with a semicolon. By pressing *return* without a semicolon, you can split a long command across multiple lines, since the command won't be executed until it is ended with a `;` followed by *return*. This multi-line format is often used for displaying and entering SQL commands, making them more readable than if they were all on a single long line. Another convention is that special words like functions and built-in variables are usually presented in `ALL CAPS` in SQL, though it is not strictly required for all database systems.

[2]Just as the shell `root` user is the most powerful superuser, with privileges to modify any file, the MySQL `root` user has permission to change any aspect of any MySQL database on the computer.

When entering commands into `mysql`, if you make a mistake, you may be tempted to type ctrl C to terminate that command and start over, like you can in the `bash` shell. Don't do it. At the `mysql>` prompt, ctrl C will terminate the whole `mysql` program. Instead, type `\c` at the end of the text you have already entered, and press return. This will end that line of input, or that continuing command, without causing it to be operated.

In the example above, `SHOW DATABASES;` is the first SQL command issued. The results of the command are presented in a simple table below the command, along with information on how long it took to carry out the operation. SQL uses plain English words for many statements, so this command is easy to understand: you are asking for a list of the databases managed by the server. One server can have many user-created databases for different projects; for example, different lab groups can have separate databases on the same server (see Figure 15.2).

The output of this command shows that there are three databases on the server: `information_schema`, `mysql`, and `test`. The first two of these databases are used by the server itself to store configurations, and should not be modified. The last database is an empty test database for checking to make sure the system is functioning, as you just did. Finally, the `EXIT;` command closes the connection between the `mysql` program and the server and returns control to the shell prompt.

During your `mysql` session, you can type `HELP;` by itself to get a list of general commands and topics, or `HELP` followed by a command name to get more specific information on that operation.

To get another view of the process as you work, within MySQL Workbench, double-click the `localhost` connection that you created earlier. This will open a new SQL Editor tab, and on the left side of the window you will see the test database (Figure 15.3). The other configuration databases are hidden from view. As you work with databases, you can enter SQL commands at the command line and monitor your progress with this graphical view. It much like navigating your filesystem at the command line and using the Finder as a graphical interface to the same files.

Creating a database and tables

So far you have connected to your local database server and looked around a bit. (Not much is there right out of the box.) Over the course of the following pages you will create a database and load in the data from several different files. These data are of a couple different types, but all were collected as part of the same project. You have already worked with one of the files, the geographical coordinates where several specimens of the deep-sea siphonophore *Marrus claudanielis* were collected with remotely operated underwater vehicles. In addition to loading these specimen data into the database, you will also load data from the CTD (conductivity, temperature, depth) instrument which collected environmental information during the dives. These two tables together will allow you to extract environmental data for the locations where the specimens were collected. This is an example of a project that includes multiple types of data that would be difficult to store efficiently in a single text file or spreadsheet.

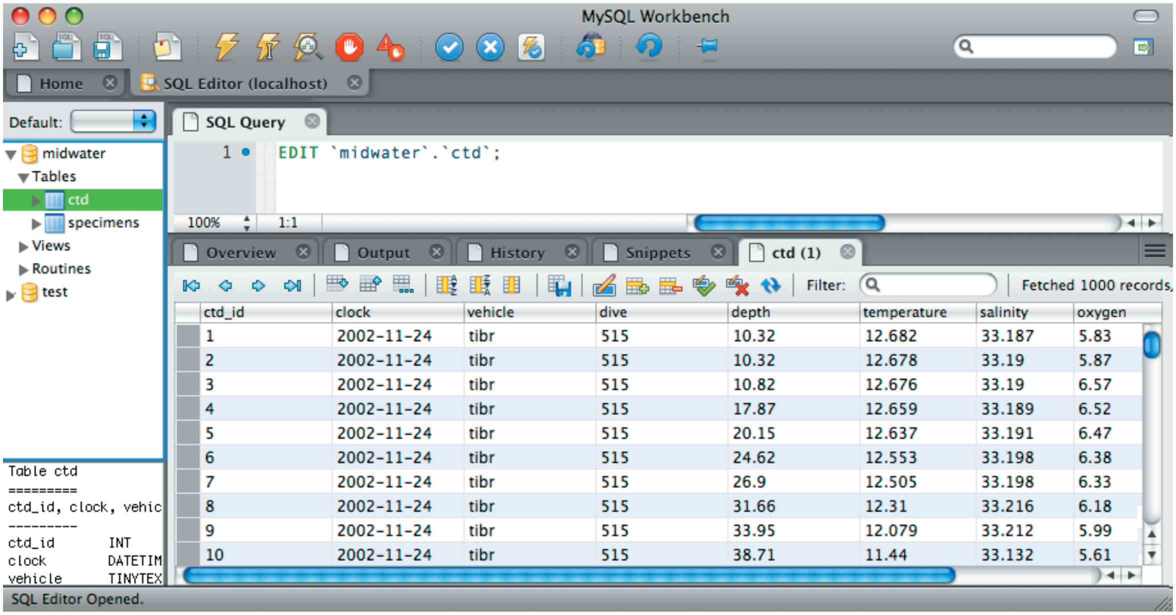


FIGURE 15.3 The MySQL Workbench GUI, viewing the contents of a table that you will create in the course of this chapter Alternative SQL database GUIs will provide similar views and options for managing and querying your database tables.

Creating and selecting the database When starting out with a project, the first step is to create an empty database using the `CREATE DATABASE` command. In the terminal window, type the following:

```
lucy$ mysql -u root

mysql> CREATE DATABASE midwater;
Query OK, 1 row affected (0.09 sec)

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| midwater           |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)

mysql>
```

You can see that there are now four databases, including the newly created `midwater` database. If you have the `localhost` connection open in MySQL Workbench, click the refresh button, the circular arrow, in the Overview pane. You will also see the database appear there too.

Since the server is responsible for several databases, you need to specify which one you want to work with. The `USE` command selects a database:

```
mysql> USE midwater;
Database changed
mysql>
```

When you issue further commands during this session, the MySQL server will know that they apply to the `midwater` database. You can switch to another database at any time by issuing another `USE` command.

Creating the `specimens` table After creating an empty database and selecting it with `USE`, the next step is to start to create tables. In a database, tables are two-dimensional data organization units, somewhat equivalent to a spreadsheet. You can create a table that has no data records in it, but there has to be at least one column (also called a field) to begin with. You can always add and remove columns later, but it is best to anticipate the needs of the table and create them all at the start. The first table you will create is the `specimens` table, into which you will load the data from the `Marrus_claudanielis.txt` file.

The first field to consider when designing a table is the primary key, the special column that contains a value which uniquely identifies each row. By convention it is best to make the first column the primary key and to use a series of unique integers as its entries. It is common practice to let the database generate a value for the primary key automatically when each row is created, so the value doesn't correspond to a value already present in the input file. After the primary key, the remaining table fields will roughly correspond to the columns of the file `Marrus_claudanielis.txt`. In your text editor, open this file from your `~/pcfb/examples` folder as you did at the start of Chapter 10. Select the Show Invisibles option in TextWrangler so that you can also inspect the white space characters:

```
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 fields of each column are delimited with a tab (which in TextWrangler is displayed as the ∆ symbol). This will be important to know when parsing the data. There is a header line that describes what each column of data contains; this is helpful for understanding the structure of the file, but will need to be skipped when parsing it.

There are six columns of data. The first column, `Dive`, contains a string that specifies both the name of the submersible vehicle (e.g., `Tiburon`) and the number of the dive (e.g., `596`). You will split this into a string and an integer, and then store each value separately in the database. `Date` is presented as a string, but we will parse it into a special date type.[3] `Lat` and `Lon` contain the latitude and longitude as strings with degrees, minutes, and compass direction. As discussed in Chapter 10, this isn't a very convenient format for storing and analyzing position information, and again we will convert it to decimal degrees so that it can be stored as a float. After `Lat` and `Lon` comes `Depth`, which in the examined part of the file contains integers. The depth, however, is a continuous measurement and it is possible that some files might be floating point numbers. You will therefore convert it to a float and store it that way. Finally, there is a `Notes` column with a string that can contain spaces and punctuation characters.

You now know that the table needs to contain the following columns:

Column name	Type
`specimen_id`	`INTEGER` ← This will be the primary key
`vehicle`	`TINYTEXT`
`dive`	`INTEGER`
`date`	`DATE`
`lat`	`FLOAT`
`lon`	`FLOAT`
`depth`	`FLOAT`
`notes`	`TEXT`

There are a few things to note above. The column names are descriptive; it is just as important to label the parts of your database with informative names as it is to give variables meaningful names when programming. By convention, the primary key for a table is often given a name similar to that of the table itself, combined with the suffix `_id`. The `vehicle` column has type `TINYTEXT` rather than type `TEXT`. This will make the database more memory-efficient, and it won't cause problems so long as you know that the names of the vehicles will never be longer than 255 characters. `TEXT` is used for the `notes` column since there is a reasonable chance a note will exceed 255 characters.

Now that you have gathered this information together, you can begin to construct the SQL command that will create a table with these columns. The com-

[3] It is probably not good practice to use the name `date`, because that is the name of a data type in SQL, but it will not cause problems in this case.

mand for creating a table is, unsurprisingly, `CREATE TABLE`. The following is the full command for constructing the table as designed above. (Note that you don't type `->`. This is part of the prompt when a command is continued across multiple lines.) You can also find the commands from this chapter in the file `mysql_commands.txt` for copying and pasting:

```
mysql> CREATE TABLE specimens (
    -> specimen_id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> vehicle TINYTEXT,
    -> dive INTEGER,
    -> date DATE,
    -> lat FLOAT,
    -> lon FLOAT,
    -> depth FLOAT,
    -> notes TEXT
    -> );
Query OK, 0 rows affected (0.10 sec)

mysql> SHOW TABLES;
+---------------------+
| Tables_in_midwater  |
+---------------------+
| specimens           |
+---------------------+
1 row in set (0.00 sec)
```

Here the `CREATE TABLE` command is spread across multiple lines and isn't executed until return is pressed after typing `;`. This command could also be entered in a single line. A one-line approach is good for scripts and automating data entry, but it makes the commands less readable and more error-prone when operating at the prompt.

`CREATE TABLE` is followed by the name of the table you want to create, and then, in parentheses, information about each column, separated by commas. That information includes the name of the field, its type, and optionally some additional information. The only column with additional parameters in this example is the primary key `specimen_id`. The statement `NOT NULL` indicates that no row can be missing a value for this column, so creating a row without a value for this field would result in an error. The database would work without this option, but including it makes the table more robust. Since `NOT NULL` isn't specified for the other fields, they can have missing data. `AUTO_INCREMENT` specifies that a new unique integer will be automatically placed in this column each time a row is added, the value of which will be one higher than the previous row that was added. This ensures that the rows have unique primary key values, and takes care of creating this value in the background so you don't have to worry about it when adding data.

`PRIMARY KEY` specifies that this column is the primary key. Some variant of this first field definition will probably be used in most of your table creation commands.

After the `CREATE TABLE` command there are a couple ways to take stock of the changes that were made to the database. `SHOW TABLES` simply gives a list of the tables available in the active database. The `DESCRIBE` command, followed by a table name, gives a summary of the structure of a table. An inspection of the results of this command confirms that the table was created as expected. Just as the shell command `ls` is indispensable for navigating your filesystem, `SHOW` and `DESCRIBE` are quick tools to get your bearings as you navigate a database:

```
mysql> DESCRIBE specimens;
+-------------+----------+------+-----+---------+----------------+
| Field       | Type     | Null | Key | Default | Extra          |
+-------------+----------+------+-----+---------+----------------+
| specimen_id | int(11)  | NO   | PRI | NULL    | auto_increment |
| vehicle     | tinytext | YES  |     | NULL    |                |
| dive        | int(11)  | YES  |     | NULL    |                |
| date        | date     | YES  |     | NULL    |                |
| lat         | float    | YES  |     | NULL    |                |
| lon         | float    | YES  |     | NULL    |                |
| depth       | float    | YES  |     | NULL    |                |
| notes       | text     | YES  |     | NULL    |                |
+-------------+----------+------+-----+---------+----------------+
8 rows in set (0.01 sec)

mysql>
```

Adding rows of data to tables and displaying table contents

You have now created a database and an empty table and are ready to add data. Before we explore tools for importing datasets from a file, it is important to understand how to add data one row at a time with SQL commands. A new row of data is added with the `INSERT` command. Here is a test command to try out; you can either copy it from the `mysql_commands.txt` file or type it in:

```
mysql> INSERT INTO specimens SET
    -> vehicle='Tiburon',
    -> dive=596,
    -> date='2003-07-03',
    -> lat=36.602,
    -> lon=-122.375,
    -> depth=1190,
    -> notes='holotype';
Query OK, 1 row affected (0.01 sec)
```

The latitude and longitude have been converted to float values by hand. The date string was reformatted to be consistent with the DATE type. The INSERT command itself is straightforward. At a minimum you need to specify the table that you are inserting the data into (here, specimens). SET is then followed by a list of column names and the values to be stored in each of them. As with the CREATE command, these are separated by commas, and they can all come on one line or in sequential lines. There are several ways to specify new data to be added to a table with the INSERT command, including some shorter formats that don't require you to enter all the column names. Specifying the column names helps avoid errors in which values get offset into another field, and it also makes your commands more readable.

To examine the contents of your table thus far, use the SELECT command with the following syntax:

```
mysql> SELECT * FROM specimens;
 +-------------+---------+------+------------+--------+----------+-------+----------+
 | specimen_id | vehicle | dive | date       | lat    | lon      | depth | notes    |
 +-------------+---------+------+------------+--------+----------+-------+----------+
 |           1 | Tiburon |  596 | 2003-07-03 | 36.602 | -122.375 |  1190 | holotype |
 +-------------+---------+------+------------+--------+----------+-------+----------+
1 row in set (0.00 sec)
```

SELECT is a very powerful command, and this simple use of it gives little indication of its potential. It is the SELECT command that will later allow us to combine data across tables and to look at specific subsets of data. Here, though, the * indicates that it should show all columns of data in the table, and the lack of other clauses for refining the search leads it to display all the rows in the table. (So far you have just inserted the one row.)

You can also view your new table and data from the MySQL Workbench. After connecting to the localhost MySQL server, select the midwater database from the Overview tab. There you will see a list of available tables. Double click on specimens to view its contents. A two-dimensional grid will appear, displaying each row and column of the table (see Figure 15.3). There are buttons for adding and deleting rows, and you can click on cells to directly edit the data. (If you edit any cells, you must click the Apply changes to data button, the one with a green check mark, for them to take effect.) This can be convenient for getting an overview of the database and making minor modifications, but use caution during database interactions, because there often is not an Undo button.

Go ahead and delete the test row you created. Either highlight it in MySQL Workbench and click the Delete row button, or issue the command DELETE FROM specimens; from within the mysql command-line interface. Be very careful with the DELETE command. As you can see it is very easy to delete all the data in a table with just a few words, leaving only the empty table behind. Like the SELECT command, DELETE assumes you want to operate on all the rows unless you specify details on which subset of rows it should consider.

Interacting with MySQL from Python

The command-line program mysql and the graphical MySQL Workbench are only two of many possible ways of interacting with the MySQL database server; you can also automate interactions with your database in a variety of ways. In the present example, the data in the Marrus_claudanielis.txt file require some parsing and calculation before they are loaded into the database. Python is a convenient tool for such manipulations, and it can interact directly with the database once the conversions are done. In fact, you already wrote a program, latlon.py, in Chapter 10 that made some of these manipulations on the exact same file. Here you will repurpose that program to write data to the MySQL database rather than to an output file. The first step is to remove unneeded parts from the script and take care of the other text reformatting issues, so that each field of data is in the correct format for loading into the database. Next, an SQL statement is built that will insert these data into the database. This is simply a string of text formatted exactly like as a command that would be entered at the mysql> prompt. Once the command is properly generated, you will use a module that connects to the MySQL server from within Python to execute this statement and add the data.

Parsing the input text

The final version of the previous script, latlon_5.py, will serve as the starting point for our new script, mylatlon.py. The first step is to strip the old script down, removing the now-unneeded code for writing the KML file. This stripped-down version of the script, along with a line for printing the parsed data to the screen, is saved as mylatlon_1.py. (We don't show the program here since it is so similar to what was presented in Chapter 10, but you can look at the code for the program in the scripts folder.) Here is the output produced when mylatlon_1.py is run; note that the path to the Marrus_claudanielis.txt file isn't specified in the program, so you need to be in the same directory as that file to run it:

```
host:ctd lucy$ mylatlon_1.py
Tiburon 596 19-Jul-03 36.602 -122.3746667 1190 holotype
JSL II 1411 16-Sep-86 39.94 -70.23833333 518 paratype
JSL II 930 18-Aug-84 40.08383333 -69.05016667 686 Youngbluth (1989)
Ventana 1575 11-Mar-99 36.704 -122.042 767
Ventana 1777 16-Jun-00 36.71 -122.045 934
Ventana 2243 9-Sep-02 36.708 -122.064 1001
Tiburon 515 24-Nov-02 36.7 -122.033 1156
Tiburon 531 13-Mar-03 24.317 -109.203 1144
Tiburon 547 31-Mar-03 24.234 -109.667 1126
JSL II 3457 26-Sep-03 40.29617 -68.1113333 862 Francesc Pages (pers.comm)
```

The initial `mylatlon_1.py` script reads the input file, skips the header line, converts the latitude and longitude to decimal degrees, and writes all the data to the screen. This takes care of most of the file parsing, but there are still two text reformatting issues that must be addressed. First, the `Dive` variable needs to be divided into the vehicle name and dive number. This will be done with a regular expression (keeping in mind that the vehicle name may have a space in it). Second, the date needs to be converted from the format found in the file to that expected by the MySQL `DATE` type (`11-Mar-99` becomes `1999-03-11`).

The date conversion could be done from scratch, but it would require a dictionary to convert the abbreviated month names to month numbers and implement rules for when to add 19 to the start of an abbreviated year (e.g., for 1999) and when to add 20 (e.g., for 2003). Fortunately, the built-in Python `datetime` module can handle all these conversions already. The `datetime` module has a `datetime` class for storing dates and times. This `datetime` class has a method called `.strptime()` that can parse `datetime` data from a string according to a specified format. It also has another method called `.strftime()` that can create a string in a specified format from a `datetime` variable. There are many formatting options, which are described at `docs.python.org/library/datetime.html`. The formatting characters used here are `%d` for day, `%b` for abbreviated month name, `%y` for two-digit year (without the century), `%Y` for the full four-digit year, and `%m` for the numeric representation of the month. In addition to these changes to how the program parses text, you will also change the type of the `Depth` record from a string to a float.

SOLVING A PROBLEM IN MORE THAN ONE WAY As it happens, SQL has its own `STR_TO_DATE()` function that operates almost identically to the Python `.strptime()` method, so you could also do a conversion as part of the SQL command when it is entered. There are usually several ways to solve a given problem, and although we chose to do the conversion in Python this time, you should investigate the range of data-handling options available to you in SQL.

To start implementing these changes, add the following line below the other `import` commands, near the top of the script:

```
from datetime import datetime
```

Note that you are importing a `datetime` class from a module called `datetime`. These are two different objects, one nested within the other. Using the same name for nested objects like this is confusing, and it should be avoided in your own code. The loop for parsing the date should be reorganized as follows:

```
# Loop over each line in the file
for Line in InFile:
   # Check line number, process if 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 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']

        Dive    = ElementList[0] # includes vehicle and dive number
        Date    = ElementList[1]
        Depth   = float(ElementList[4])
        Comment = ElementList[5]

        LatDegrees = decimalat(ElementList[2])
        LonDegrees = decimalat(ElementList[3])

        # NEW CODE ADDED BELOW HERE
        #Isolate the vehicle and dive number from the Dive field
        SearchStr='(.+?) (\d+)'
        Result = re.search(SearchStr, Dive)
        Vehicle = Result.group(1)
        DiveNum = int(Result.group(2))

        # Reformat date
        # Create a datetime object from a string
        DateParsed = datetime.strptime(Date, "%d-%b-%y")
        DateOut = DateParsed.strftime("%Y-%m-%d") # string from datetime object

        print Vehicle, DiveNum, DateOut, LatDegrees, LonDegrees, Depth, Comment

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

Run the script to confirm that you get a reformatted date beginning with the four-digit year, and that the dive number is parsed correctly.

Formulating SQL from the data

Now that each data field has been parsed from the file and all fields formatted appropriately, these data can be packaged for insertion into the database. Each line of data will be added to the database with an `INSERT INTO` command, just like the one from the SQL example. This SQL command is just a string that describes what you want to do with some data. Before the final code for connecting to the database is added to the program, you will print the SQL command to the screen. This is a good step to take with any program that can modify a database. It allows you to catch potential problems before difficult-to-fix database mistakes are made.

A simple way to build up a large string with many fields is with the `%` string formatting operator, introduced in Chapter 8. Triple quotes are used so that the string can be split across multiple lines for readability. Comment out the existing `print` line and add the code for creating the SQL statement immediately below it:

```
   # print Vehicle, DiveNum, DateOut, LatDegrees, LonDegrees, Depth, Comment
   SQL = """INSERT INTO specimens SET
vehicle='%s',
dive=%d,
date='%s',
lat=%.4f,
lon=%.4f,
depth=%.1f,
notes='%s' ;
""" % (Vehicle, DiveNum, DateOut, LatDegrees, LonDegrees, Depth, Comment)

   print SQL
```

Notice that the SQL command we are generating requires quotation marks around strings, and therefore single quotes are used within the triple-quoted string. This version of the program is saved as `mylatlon_3.py`. The output of the program is now a series of SQL commands, the first two of which are shown here:

```
host:ctd lucy$ mylatlon_3.py
INSERT INTO specimens SET
      vehicle='Tiburon',
      dive=596,
      date='2003-07-19',
      lat=36.6020,
      lon=-122.3747,
      depth=1190.0,
      notes='holotype' ;

INSERT INTO specimens SET
      vehicle='JSL II',
      dive=1411,
      date='1986-09-16',
      lat=39.9400,
      lon=-70.2383,
      depth=518.0,
      notes='paratype' ;
```

These SQL commands will enter the specimen data into the database row by row. You could even copy and paste one of these commands right into an open `mysql` session to enter the data. (The semicolons are optional when commands are submitted through Python, but we have included them here so you can paste the output directly at a `mysql>` prompt.)

Executing SQL commands from Python

All that remains is to connect to the MySQL server from within Python and execute the commands. There are a few steps to this.

Installing the `MySQLdb` Python module There are quite a few things that have to happen behind the scenes to make a connection to the MySQL database. Fortunately, there is a Python module called `MySQLdb` which takes care of all the work under the hood. The most complicated part of connecting to MySQL from Python is installing this module, but you only have to do it once. `MySQLdb` has some known installation issues with older versions of OS X, MySQL, and Python. If you encounter any errors or something doesn't seem consistent with the instructions provided here, consult Chapter 21, search the Web for the error message you get, or visit `practicalcomputing.org` for additional guidance. If you are running Ubuntu Linux, `MySQLdb` can be installed with the Synaptic Package Manager. Note that you must install MySQL first before installing the module.

If you are running OS X, download the `MySQLdb` module from `sourceforge.net/projects/mysql-python/`. Double-click the archive to uncompress and expand it.[4] Open a terminal window, type `cd` space and drag the icon for the folder you just expanded to the terminal window, and press return to move into that directory. Read the installation instructions in the `README` file, and then proceed with the installation:

```
host:ctd lucy$ cd ~/Downloads/MySQL-python-1.2.3/
host:MySQL lucy$ cat README
host:MySQL lucy$ python setup.py build
host:MySQL lucy$ sudo python setup.py install
```

The names of the files and folders may differ slightly if you download a later version of `MySQLdb`. The `cat README` command will display information about the installation process in the supplied `README` file. If the suggested installation commands differ from the `python` and `sudo` commands shown here, then follow the instructions in the `README` file instead.

It is a good idea to test that the module loads correctly using the interactive Python prompt before giving it a try in a program. Move to the `examples/ctd` folder, and try to load the module in Python:

```
host:MySQL lucy$ cd ~/pcfb/examples/ctd
host:ctd lucy$ python
Python 2.6.1 (r261:67515, Feb 11 2010, 00:51:29)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import MySQLdb
>>>
```

[4] For more information about special cases of unarchiving and installing software, see Chapter 21.

In this example, there were no warnings or errors. Even if you get some warnings, the module may still work fine. Restarting your computer may help resolve some errors.

Establishing the database connection Once you have installed `MySQLdb` on your computer, you still need to import it into your Python program to use it. Add the following line below the `import` statements that are already at the top of your script:

```
import MySQLdb
```

Next, you need to create a connection to the MySQL database. You can create a single connection to the database near the top of your program, and then use and reuse it wherever you like. Place the following lines right before the `for` loop:

```
MyConnection = MySQLdb.connect( host = "localhost", \
user= "root", passwd = "", db = "midwater")
MyCursor = MyConnection.cursor()
```

The first of these lines (shown here split into two with a \ to escape the line ending) creates the actual connection object, `MyConnection`. It needs information about the network address of the MySQL server (`localhost` since you are running the server right on your computer), username, password (left blank here since we haven't created one), and the database on the server that you want to connect to. Once you have the connection object, you use it to create a cursor object. You can think of this database cursor like the cursor at the command line: it is the point at which you interact with the MySQL program. This cursor is used to submit commands and retrieve results.

At the very end of the program, add two lines to close the cursor and database connection:

```
MyCursor.close()
MyConnection.close()
```

Executing SQL commands At this point, you have generated SQL command strings and have a connection to the database. All you need to do is execute the SQL commands so that the program adds the data to the database. Now that you have taken care of all the housekeeping, the actual execution command is only a single line. Add it right after the line that prints the SQL variable to the screen:

```
MyCursor.execute(SQL)
```

This line executes the `SQL` string you created earlier, using the database cursor you opened before the loop. Each time through the loop, it executes a new SQL command and adds another row of data to the database.

The final program, saved as `mylatlon_4.py`, is below:

```
#! /usr/bin/env python
"""
mylatlon_4.py
import latitude longitude records from a text file,
format them into a SQL command, and enter the records into a database
"""

import re # Load regular expression module
from datetime import datetime # Load datetime class from the datetime module
import MySQLdb

# 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 definition

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

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

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

# Create the database connection
# Often you will want to use a variable instead of a fixed string
# for the database name

MyConnection = MySQLdb.connect( host = "localhost", user = "root", \
      passwd = "", db = "midwater")
MyCursor = MyConnection.cursor()
```

```
# Loop over each line in the file
for Line in InFile:
   # Check the line number, process if 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 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']

     Dive    = ElementList[0] # includes vehicle and dive number
     Date    = ElementList[1]
     Depth   = float(ElementList[4])
     Comment = ElementList[5]
     LatDegrees = decimalat(ElementList[2])
     LonDegrees = decimalat(ElementList[3])

     #Isolate the vehicle and dive number from the Dive field

     SearchStr='(.+?) (\d+)'
     Result = re.search(SearchStr, Dive)
     Vehicle = Result.group(1)
     DiveNum = int(Result.group(2))

     #Reformat date
     # Create a datetime object from a string
     DateParsed = datetime.strptime(Date, "%d-%b-%y")
     # Create a string from a datetime object
     DateOut = DateParsed.strftime("%Y-%m-%d")
     #print Vehicle, DiveNum, DateOut, LatDegrees, LonDegrees, Depth, Comment
     SQL = """INSERT INTO specimens SET
  vehicle='%s',
  dive=%d,
  date='%s',
  lat=%.4f,
  lon=%.4f,
  depth=%.1f,
  notes='%s' ;
""" % (Vehicle, DiveNum, DateOut, LatDegrees, LonDegrees, Depth, Comment)
     print SQL
     MyCursor.execute(SQL)
   LineNumber += 1 # This is outside the if, but inside the for loop

# Close the files
InFile.close()
MyCursor.close()
MyConnection.close()
```

From the folder that contains the Marrus_claudanielis.txt file, execute mylatlon_4.py. The output will look the same as the output of mylatlon_3.py, but behind the scenes the data are being added to the database! (Avoid re-running this command during testing or it will add duplicate records to your database.) From within the mysql command-line interface, use the SELECT command to take a look again at the contents of the specimens table. If you are starting a new mysql session, remember to first select the midwater database with the USE midwater; command. The output below has been edited slightly to fit on the page:

```
mysql> SELECT * FROM specimens;
+------------+---------+------+------------+--------+----------+------+--------------------+
|specimen_id | vehicle | dive | date       | lat    | lon      |depth | notes              |
+------------+---------+------+------------+--------+----------+------+--------------------+
|          4 | Tiburon |  596 | 2003-07-19 | 36.602 | -122.375 | 1190 | holotype           |
|          5 | JSL II  | 1411 | 1986-09-16 |  39.94 | -70.2383 |  518 | paratype           |
|          6 | JSL II  |  930 | 1984-08-18 | 40.084 | -69.0502 |  686 | Youngbluth (1989)  |
|          7 | Ventana | 1575 | 1999-03-11 | 36.704 | -122.042 |  767 |                    |
|          8 | Ventana | 1777 | 2000-06-16 |  36.71 | -122.045 |  934 |                    |
|          9 | Ventana | 2243 | 2002-09-09 | 36.708 | -122.064 | 1001 |                    |
|         10 | Tiburon |  515 | 2002-11-24 |   36.7 | -122.033 | 1156 |                    |
|         11 | Tiburon |  531 | 2003-03-13 | 24.317 | -109.203 | 1144 |                    |
|         12 | Tiburon |  547 | 2003-03-31 | 24.234 | -109.667 | 1126 |                    |
|         13 | JSL II  | 3457 | 2003-09-26 | 40.296 | -68.1113 |  862 | Pages (pers.comm)  |
+------------+---------+------+------------+--------+----------+------+--------------------+
10 rows in set (0.00 sec)
```

You can also inspect the modified table from the MySQL Workbench graphical interface. Your values for specimen_id might vary from those shown here, because the AUTO_INCREMENT counter keeps track of all the rows that have ever been added even if they have subsequently been removed.

Bulk-importing text files into a table

Typically, when starting to work with a database, you will already have your data stored in text files or spreadsheets, which you want to import into database tables. In the next component of this example you will load a new data table named ctd with environmental data measured from the CTD sensors on the submarines that collected six of these specimens. Unlike the specimens data, the columns of the files correspond exactly to the columns of the table that you create, and no conversion is needed. While you could import these data into the database with another custom Python program, because they are already formatted, you can add the data with simple SQL commands. Before you can add any data, though, you will need to create the table.

Creating the `ctd` *table*

Open a new terminal window to get a shell prompt, and change into the `~/pcfb/examples/ctd/` directory. Then, generate a list of the files that start with the word `Marrus` and view the header of the first file using the `head` command, which is described fully in the next chapter:

```
host:~ lucy$ cd ~/pcfb/examples/ctd
host:ctd lucy$ ls Marrus*
Marrus_ctdTib515.txt    Marrus_ctdTib596.txt    Marrus_ctdVen2243.txt
Marrus_ctdTib531.txt    Marrus_ctdVen1575.txt
Marrus_ctdTib547.txt    Marrus_ctdVen1777.txt
host:ctd lucy$ head Marrus_ctdTib515.txt
rovCtdDtg,vehicle,depth,temper,salin,oxyg,lat,lon
2002-11-24 14:24:15,tibr,10.32,12.682,33.187,5.83,36.571183,-122.52263
2002-11-24 14:24:45,tibr,10.32,12.678,33.19,5.87,36.70004157,-122.03345157
2002-11-24 14:25:15,tibr,10.82,12.676,33.19,6.57,36.70000171,-122.03336828
2002-11-24 14:25:45,tibr,17.87,12.659,33.189,6.52,36.700005,-122.03334412
2002-11-24 14:26:15,tibr,20.15,12.637,33.191,6.47,36.69998512,-122.03335
...
host:ctd lucy$
```

The first things that stand out are that the values are separated by commas, and that there is a header row that describes what each of the values are. From this information you can generate and execute a `CREATE TABLE` command that has all the needed fields:

```
mysql> CREATE TABLE ctd (
    -> ctd_id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> clock DATETIME,
    -> vehicle TINYTEXT,
    -> dive INTEGER,
    -> depth FLOAT,
    -> temperature FLOAT,
    -> salinity FLOAT,
    -> oxygen FLOAT,
    -> lat FLOAT,
    -> lon FLOAT
    -> );
```

Some of these fields correspond to the fields of the `specimens` table, and this will ultimately help you link corresponding pieces of information. The `date` field in these files happens to be properly formatted for MySQL to import directly.[5] Otherwise, you might have to convert them with regular expressions, a separate program, or one of the SQL date functions like `STR_TO_DATE`.

[5] What a fortunate coincidence...

Importing data files with the `LOAD DATA` *command*

The command for loading data into a table from a text file is long but for the most part self-explanatory. Here is an example for the first CTD file:

```
LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdTib515.txt'
   INTO TABLE ctd
   FIELDS TERMINATED BY ','
   IGNORE 1 LINES
   (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)
   SET dive=515;
```

In this example, there are only two parts of this command that will change from file to file: the name of the file and the value placed into the `dive` variable, which is derived from the filename. The first line of the command specifies that it is a `LOAD DATA` statement and that the file is on the local computer, and provides the path to the file. The `INTO TABLE` portion of the statement specifies which table to load the data into. The next two lines indicate that the fields are separated by commas, and that the first line is a header line that should be skipped. The names of the table columns that the fields should be loaded into are then specified, within parentheses, in the order that they occur in the file. The last line sets the value of the dive field in the table to `515` for all the added rows. This dive number is in the filename, but isn't located within the file itself.

To generate the commands for loading data from all the files, list the `ctd` directory using the command `ls -1 Marrus*`. (The flag is the number 1). This will show a column listing just the CTD file names. Note that there are no files from the JSL II submarine, so specimens collected with that vehicle will not have corresponding temperature information. Copy the file list into a text editor and use regular expressions to modify this list into a series of one-line commands as described here. Search for the following:

```
(\w+?(\d+)\.txt)
```

Then replace all with the text below,[6] which you can copy from the file `mysql_commands.txt`:

```
LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/\1'
INTO TABLE ctd
FIELDS TERMINATED BY ',' IGNORE 1 LINES
(clock,vehicle,depth,temperature, salinity,oxygen,lat,lon)
SET dive=\2;
```

---

[6] This search is a bit tricky because it uses nested parentheses to capture replacement text. The text in the outermost pair, including the dive number, is saved as `\1`. The inner parentheses save the dive number alone as `\2`.

The result of this replacement will be a series of commands, shown below, which will load each file into your database.

The full set of additional commands is also saved in the `mysql_commands.txt` example file. You could also type the command once, and then recycle it while replacing the file name on the first line and dive number on the last line. As with the `bash` shell, you can use the ↑ key to move back through your command history to edit the names. Another very useful aspect of the `mysql>` prompt is that tab will also auto-complete `mysql` commands and variable names that are known to the database. Note that if you paste all of these lines at once, it may overwhelm the terminal program's buffer, and some of them may get garbled. On our computers, pasting four commands at a time worked without any problems.

```
mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdTib515.txt'
   INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
   (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=515;

mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdTib531.txt'
   INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
   (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=531;

mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdTib547.txt'
   INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
   (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=547;

mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdTib596.txt'
   INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
   (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=596;

mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdVen1575.txt'
   INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
   (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=1575;

mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdVen1777.txt'
  INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
  (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=1777;

mysql> LOAD DATA LOCAL INFILE '~/pcfb/examples/ctd/Marrus_ctdVen2243.txt'
  INTO TABLE ctd  FIELDS TERMINATED BY ','  IGNORE 1 LINES
  (clock,vehicle,depth,temperature,salinity,oxygen,lat,lon)  SET dive=2243;

mysql>
```

After these commands have run, all of the CTD data have been loaded from files into the `midwater` database's `ctd` table. We will explore this table in a later section.

Other approaches to automating the import of files into a database are postulated at the end of the chapter.

Exporting and importing databases as SQL files

It is common to distribute databases as SQL files. These are text files with all the commands needed to create tables (and sometimes the database itself as well), and to add all the rows of data to the tables. This is achieved with the `mysqldump` command, which is also a convenient way to backup a database. Not only are the data themselves preserved, so is the structure of the database.

We have provided both tables of the database created above as a SQL file called `midwater.sql`, also available in the `ctd` folder. It was created with the shell command:

```
host:ctd lucy$ mysqldump -u root midwater > midwater.sql
```

(Note that this is a `bash` command, not a `mysql` command.) Open up the `midwater.sql` file in a text editor. Some of the commands in the file will be familiar, whereas some use statements you have seen but in different formulations, and some aren't covered in this book.

If you are unable to create and load the database as described in previous sections but would still like to follow along with the data-mining examples below, you can load the database from the `midwater.sql` file. First create the empty `midwater` database in `mysql`, and then at the shell enter the following command to execute the SQL commands in the file:

```
host:~ lucy$ mysql -u root midwater < ~/pcfb/examples/ctd/midwater.sql
```

This general strategy works for executing any set of SQL statements; they don't have to be commands for creating and filling tables.[7] Because SQL commands remain largely the same among various implementations of relational database systems, you might be able to import this file or a slightly modified version into another database management program.

Exploring data with SQL

Now that all the data for this project are in the database, you can use SQL commands to summarize, update, and extract information.

Summarizing tables with `SELECT` *and* `COUNT`

With the small `specimens` table, we already examined all the rows using the `SELECT * FROM` specimens command. Here are some more uses of `SELECT`, and

[7] This use of the < operator has not been covered in this book, but it is a variation of the redirection operator > that you have been using to save output to a file. When used in the other direction, pointing left, it causes a file to be used as input to a program or command.

ways to refine the output. A basic question about a database is, "How many rows does my table contain?" You can use a slightly modified `SELECT` statement to get an answer:

```
mysql> SELECT COUNT(*) FROM specimens;
+----------+
| COUNT(*) |
+----------+
|       10 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM ctd;
+----------+
| COUNT(*) |
+----------+
|     3738 |
+----------+
1 row in set (0.00 sec)
```

By replacing `*` with `COUNT(*)`, you now retrieve a single row that contains the count of the number of rows retrieved by `SELECT`, rather than all the data rows themselves. The command `COUNT` is one of SQL's statistical functions, and it accepts parameters passed to it within parentheses. Other math and statistical operators are described below and summarized in Table 15.2.

It is also possible to extract data from only particular columns with `SELECT`. Instead of using `*`, which is a wildcard for all columns, you can specify the columns you want, separating them by commas:

```
mysql> SELECT vehicle,date FROM specimens;
+---------+------------+
| vehicle | date       |
+---------+------------+
| Tiburon | 2003-07-19 |
| JSL II  | 1986-09-16 |
| JSL II  | 1984-08-18 |
| Ventana | 1999-03-11 |
| Ventana | 2000-06-16 |
| Ventana | 2002-09-09 |
| Tiburon | 2002-11-24 |
| Tiburon | 2003-03-13 |
| Tiburon | 2003-03-31 |
| JSL II  | 2003-09-26 |
+---------+------------+
10 rows in set (0.00 sec)
```

Collating data with GROUP BY

A common task is to see how many distinct values a given column has. It would, for instance, be informative to know how many vehicles have records. This can be done in a couple of different ways:

```
mysql> SELECT DISTINCT vehicle FROM specimens;
+---------+
| vehicle |
+---------+
| Tiburon |
| JSL II  |
| Ventana |
+---------+
3 rows in set (0.03 sec)

mysql> SELECT vehicle,COUNT(*) FROM specimens GROUP BY vehicle;
+---------+----------+
| vehicle | COUNT(*) |
+---------+----------+
| JSL II  |        3 |
| Tiburon |        4 |
| Ventana |        3 |
+---------+----------+
3 rows in set (0.21 sec)

mysql> SELECT vehicle,dive,COUNT(*) FROM ctd GROUP BY vehicle, dive;
+---------+------+----------+
| vehicle | dive | COUNT(*) |
+---------+------+----------+
| tibr    |  515 |      491 |
| tibr    |  531 |     1348 |
| tibr    |  547 |      486 |
| tibr    |  596 |      760 |
| vnta    | 1575 |      100 |
| vnta    | 1777 |      210 |
| vnta    | 2243 |      343 |
+---------+------+----------+
7 rows in set (0.00 sec)

mysql>
```

The `SELECT DISTINCT` command is the simplest to type, but it doesn't tell you how many rows there are for each vehicle type. To get the count for each variable, use the alternative command that includes the `GROUP BY` clause. It groups the rows by shared vehicle values, and then counts the number of rows in each of these groups with `COUNT(*)`. Note that both vehicle and `COUNT(*)` are selected;

TABLE 15.2 Selected SQL math and statistical operators and functions

Function or operator	Meaning
+, –, *, /	Basic math operators
AVG	Average of the values
COUNT	Count of the values
MAX	Maximum value
MIN	Minimum value
STD	Standard deviation
SUM	Sum of the values

if only `COUNT(*)` is specified then you will get counts without knowing which vehicles they are associated with. You can use multiple columns for `GROUP BY`, so that each row in the result will be for a unique observed combination of these columns.

Mathematical operations in SQL

In addition to returning values from a table, SQL can perform mathematical and statistical operations on the data that are retrieved (see Table 15.2). To use these operators, construct a formula within parentheses, connecting field names with math symbols, as with `(depth * 3.3)`. You can also use the statistical functions by placing a field name in parentheses after the parameter name. For example, to get the average depth for dives, grouped by each vehicle, you can use the following command:

```
mysql> SELECT vehicle, AVG(depth) FROM specimens GROUP BY vehicle;
+---------+------------------+
| vehicle | avg(depth)       |
+---------+------------------+
| JSL II  | 688.666666666667 |
| Tiburon |             1154 |
| Ventana | 900.666666666667 |
+---------+------------------+
3 rows in set (0.06 sec)
```

Refining selections by row with `WHERE`

In addition to isolating particular columns, you can also isolate particular rows from a table. This is done with the `WHERE` clause (Figure 15.4). To see only the rows that pertain to the vehicle Tiburon, use the following command:

```
mysql> SELECT * FROM specimens WHERE vehicle='Tiburon';
+-------------+---------+------+------------+--------+----------+-------+----------+
| specimen_id | vehicle | dive | date       | lat    | lon      | depth | notes    |
+-------------+---------+------+------------+--------+----------+-------+----------+
|           4 | Tiburon |  596 | 2003-07-19 | 36.602 | -122.375 |  1190 | holotype |
|          10 | Tiburon |  515 | 2002-11-24 |   36.7 | -122.033 |  1156 |          |
|          11 | Tiburon |  531 | 2003-03-13 | 24.317 | -109.203 |  1144 |          |
|          12 | Tiburon |  547 | 2003-03-31 | 24.234 | -109.667 |  1126 |          |
+-------------+---------+------+------------+--------+----------+-------+----------+
4 rows in set (0.00 sec)
```

Note that the equality operator in SQL is a single = sign, not == as in Python and many other languages. You can slice data by combined criteria that pertain to columns (`dive, data`) and rows (`WHERE vehicle = 'Tiburon'`) together:

```
mysql> SELECT dive,date FROM specimens WHERE vehicle='Ventana';
+------+------------+
| dive | date       |
+------+------------+
|  596 | 2003-07-19 |
|  515 | 2002-11-24 |
|  531 | 2003-03-13 |
|  547 | 2003-03-31 |
+------+------------+
```

The `WHERE` phrase is very adaptable, and you can use it with approximate matches using `LIKE` or even regular expressions with `REGEXP`. Instead of testing for an exact match to a string with `=`, you can use `LIKE` to retrieve matches to a portion of the string. `LIKE` uses `%` as a wildcard the same way that `*` is used in the `bash` shell and how `.*` is used in regular expressions:

```
mysql> SELECT vehicle,dive FROM specimens WHERE vehicle LIKE 'TIB%';
+---------+------+
| vehicle | dive |
+---------+------+
| Tiburon |  596 |
| Tiburon |  515 |
| Tiburon |  531 |
| Tiburon |  547 |
+---------+------+
```

To perform a regular expression search, you use `WHERE` *`field`* `REGEXP` *`query`*, and *`query`* is a string containing the search term. Any field values which match the regular expression will be returned. The SQL regular expression syntax matches most closely with the terms used in the `bash` shell (see Appendix 2), and it doesn't include all of the wildcards like `\w` and `\d`. You are able to specify the beginning and ending of strings with `^` and `$`, any character with a period, and range of characters with square brackets `[A-Z]`:

```
mysql> SELECT vehicle,dive FROM specimens WHERE vehicle REGEXP '^V';
+---------+------+
| vehicle | dive |
+---------+------+
| Ventana | 1575 |
| Ventana | 1777 |
| Ventana | 2243 |
+---------+------+
```

Expressions built with WHERE also commonly use numerical comparison operators and logical statements, just like if statements in Python. This will probably be one of your most common uses of a SELECT statement. For example:

```
mysql> SELECT vehicle,dive FROM specimens WHERE dive < 1000;
+---------+------+
| vehicle | dive |
+---------+------+
| Tiburon |  596 |
| Tiburon |  515 |
| Tiburon |  531 |
| Tiburon |  547 |
| JSL II  |  930 |
+---------+------+
```

If you build up a logical sequence of comparisons, be sure to think out the use of AND and OR. Two comparisons linked by an OR will return the merged set of values where those tests are true. For example, to return the combined set of dive numbers for Tiburon and JSL II, you could use:

```
SELECT vehicle, dive from specimens
  WHERE vehicle LIKE "Tib%" OR vehicle LIKE "JSL%";
```

If you wanted records from Tiburon and JSL, but tried using an AND statement, you would get no results. We will use WHERE with AND to return a desired subset of the CTD records in a later example.

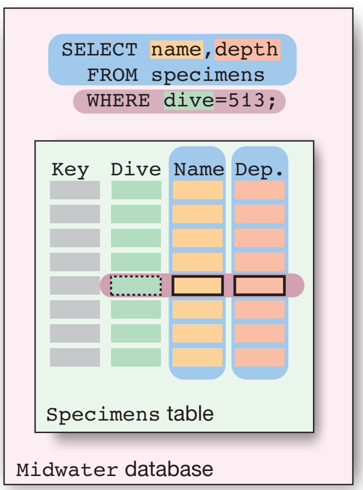


FIGURE 15.4 A graphical view of extracting data from a database The SELECT command starts by defining the columns (blue fields) to retrieve. The WHERE command (maroon stripe) refines which rows of these columns should be extracted, based on values in those or other columns.

Modifying rows with UPDATE

Once data are loaded into a database you will often want to modify them. In this database, different names have been used for the vehicles in the `specimen` and `ctd` files:

```
mysql> SELECT vehicle,COUNT(*) FROM specimens GROUP BY vehicle;
+---------+----------+
| vehicle | COUNT(*) |
+---------+----------+
| JSL II  |        3 |
| Tiburon |        4 |
| Ventana |        3 |
+---------+----------+
3 rows in set (0.01 sec)

mysql> SELECT vehicle,COUNT(*) FROM ctd GROUP BY vehicle;
+---------+----------+
| vehicle | COUNT(*) |
+---------+----------+
| tibr    |     3085 |
| vnta    |      653 |
+---------+----------+
2 rows in set (0.09 sec)
```

This is less than desirable. Below are `UPDATE` commands that change the vehicle abbreviations in the `ctd` table to match the full vehicle names used in the `specimens` table:

```
mysql> UPDATE ctd SET vehicle='TIBURON' WHERE vehicle='tibr';
Query OK, 3085 rows affected (0.07 sec)
Rows matched: 3085  Changed: 3085  Warnings: 0

mysql> UPDATE ctd SET vehicle='VENTANA' WHERE vehicle='vnta';
Query OK, 653 rows affected (0.05 sec)
Rows matched: 653  Changed: 653  Warnings: 0
```

Here the `WHERE` clause is acting just as it did for the `SELECT` command: it is restricting the command to a subset of rows where the specified criteria are true. The `SET` clause is acting as it did in the `INSERT INTO` and `LOAD DATA` from commands: it is assigning a particular value to a particular field.

The previous examples were exploring data tables, but this command is altering the data table in a way that is important for the rest of the operations in this chapter. Execute these `UPDATE` commands (also available in example file) before proceeding.

Selecting data across tables

So far there has been no interaction between the tables in databases. They have been loaded with data and analyzed independently. Combining data across tables is one of the most powerful abilities of relational databases. It is also where relational databases get their name—relationships can be defined between data across tables. This allows for complex database structures that would be very inefficient to represent with two-dimensional grids, even though each table in the database is two-dimensional.

Combining data across tables is not as complicated as you might think. In most cases it just requires modifying the `SELECT` statement so it is pulling data from multiple tables according to particular relationships. To access more than one table, you put the names of the tables after the `FROM` statement, separated by commas. Because you are now querying against two tables with different dimensions and different fields, some of which may have the same name, you need to specify which table you are talking about when you indicate a field name. This is done with a dot notation similar to some of the methods you used in Python. A particular field is specified with the name of the table, a dot, and then the name of the field. To indicate the vehicle field of the specimen table, for instance, you would use `specimen.vehicle`.

In the following series of `SELECT` commands you will extract environmental data from the `ctd` table corresponding to the collection depth of particular organisms from the `specimens` table. As a first step, just select the `vehicle`, `dive`, and `depth` fields for the holotype specimen (the specimen that was chosen as the representative of the entire species):

```
mysql> SELECT specimens.vehicle, specimens.dive, specimens.depth
    -> FROM specimens
    -> WHERE specimens.notes="holotype";
+---------+------+-------+
| vehicle | dive | depth |
+---------+------+-------+
| Tiburon |  596 |  1190 |
+---------+------+-------+
```

To list all the CTD data for this dive by this vehicle, try the following command:

```
mysql> SELECT ctd.* FROM ctd, specimens
    -> WHERE specimens.notes="holotype"
    -> AND ctd.vehicle=specimens.vehicle AND ctd.dive=specimens.dive;
```

This will return hundreds of rows of data taken during dive 596 by the remotely operated underwater vehicle Tiburon.[8] Neither the dive number or vehicle were stated explicitly, though. The portion of the CTD data displayed was restricted by

[8] To only print the first 10 rows, add `LIMIT 10` to the end of the command.

specifying that the notes field for specimens had to be "holotype", and then that the dive and vehicle fields in the ctd table had to match the corresponding values from the selected row from the specimens table.

At this point you could scroll through these data, look for the depth that is closest to that of the specimen, and extract the desired CTD data—or you could let the database system do the work for you. Depth can't be specified across tables in the same way that vehicle and dive were since the CTD measurements are taken at intervals, and the chances of getting measurements at the exact depth where the specimen was collected are slim.

The following command shows selected CTD data for the row that has the minimal depth, as specified with `MIN(ctd.depth)`, out of the depths that are greater than or equal to the collection depth, as specified by `ctd.depth >= specimens.depth`. This is one method to find the closest value to a known value. This command also displays all the vehicle and dive values from the `specimens` table for this record, showing how each row of the result can contain data that are gathered from across multiple tables:

```
mysql> SELECT specimens.vehicle, specimens.dive, MIN(ctd.depth),
    -> ctd.temperature, ctd.salinity, ctd.oxygen
    -> FROM ctd, specimens
    -> WHERE specimens.notes="holotype"
    -> AND ctd.vehicle=specimens.vehicle AND ctd.dive=specimens.dive
    -> AND ctd.depth >= specimens.depth;
+---------+------+----------------+-------+----------+--------+
| vehicle | dive | MIN(ctd.depth) | temp. | salinity | oxygen |
+---------+------+----------------+-------+----------+--------+
| Tiburon |  596 | 1190.170043945 | 3.405 |   34.284 |   0.05 |
+---------+------+----------------+-------+----------+--------+
```

Next you will add a command like this into a Python program, to retrieve a combined set of values for each specimen, not just for the holotype.

Generating output using Python

For most data extraction procedures there are ways to achieve everything with a series of SQL commands. As you saw at the beginning of this chapter, though, sometimes it is easier to process files using the familiar Python environment, generating customized SQL queries and sending them to MySQL using the `MySQLdb` module. In the next example, you will take the command derived above to extract CTD data and use a Python script to apply it to each row in the `specimen` table. The resulting output is a tab-delimited table that could quickly be formatted for use in a publication. For this script to work, you need to have generated or imported the `specimens` and `ctd` tables in the `midwater` database, and performed the `UPDATE` command to standardize vehicle names, as described in the body of this chapter.

```
#! /usr/bin/env python
"""
sqlmerge.py
using the mysql database 'midwater', with its tables 'ctd' and 'specimens',
look up the dive and depth for each specimen, and extract the corresponding
temperature, salinity, and oxygen from the ctd table

output the combined results as a tab-delimited table
"""

import re         # Load regular expression module
import MySQLdb    # must be installed separately

# Create the database connection. Often you will want to use a
# a variable to hold the database name, instead of a fixed string
MyConnection = MySQLdb.connect( host = "localhost", user = "root", \
                                passwd = "", db = "midwater")
MyCursor = MyConnection.cursor()
SQL = """SELECT specimen_id,vehicle,dive,date,depth,lat,lon from specimens;"""
SQLLen = MyCursor.execute(SQL)  # returns the number of records retrieved

# MyCursor is now "loaded" with the results of the SQL command
# AllOut will become a list of all the records selected
AllOut = MyCursor.fetchall()
# print AllOut ## Debugging
# Print the header line
print "Vehicle\tDive\tDate\tDepth\tLat.\tLong.\tTemperature\tSalinity\tOxygen"

# Step through each record and create a new SQL command to retrieve
# the corresponding values from the other DB
for Index in range(SQLLen):
   # two dimensional indexing:
   # from the Indexed record, take the first item (the primary_key)
   Spec_id = AllOut[Index][0]

   # Other ways to print debugging information
#  vehicle,dive,date,depth,lat,lon =   AllOut[Index][1:]
#  print "%s\t%d\t%s\t%.1f\t%.4f\t%.4f\t" % AllOut[Index][1:]
#  vehicle,dive,date, depth, lat,lon,

# insert spec_id (the primary key) into each command
   SQL = """SELECT MIN(ctd.depth),ctd.temperature,ctd.salinity,ctd.oxygen
   from ctd, specimens where
   specimens.specimen_id=%d and specimens.vehicle=ctd.vehicle and
   specimens.dive=ctd.dive and ctd.depth>=specimens.depth ; """ % Spec_id

   # print SQL   ## Uncomment to test the command structure before running

   SQLLen = MyCursor.execute(SQL)
   NewOut = MyCursor.fetchall()
```

```
   if SQLLen < 1 or NewOut[0][0]==None:  # Some records don't have CTD data
      print  "%s\t%d\t%s\t%.1f\t%.4f\t%.4f\t" % AllOut[Index][1:] \
         + "NaN\tNaN\tNaN"
   else:
      print  "%s\t%d\t%s\t%.1f\t%.4f\t%.4f\t" % AllOut[Index][1:] \
         + "%.2f\t%.3f\t%.2f" % NewOut[0][1:]
# Close the files
MyCursor.close()
MyConnection.close()
```

In this example script, saved as `sqlmerge.py` in the ~/pcfb/scripts folder, the results fetched from the SQL command are loaded into the variable `AllOut` using the `.fetchall()` function. This is equivalent to the file operator `.readlines()` which loads all lines of a file into a variable.

The basic approach for one of these retrievals is to execute the query and then to load the data from the `MyCursor` object:

```
MyCursor.execute(SQL)
AllOut = MyCursor.fetchall()
```

Instead of `.fetchall()` you could also use a loop and the `.fetchone()` method to retrieve one record at time from the `MyCursor` results. This would be a better approach for large datasets, as all the results aren't stored in the memory at once.

To print or otherwise access the contents of each line stored in `AllOut`, you use two indices in square brackets: the first shows which line to use and the second tells which field within that line you want to print. The order of the fields in `AllOut` correspond to their order in the first SQL command that we ran.

For instance, the second line of the `AllOut` variable would be `AllOut[1]`, corresponding to the specimen collected during Tiburon dive 515. Within this line, the fourth value `AllOut[1][3]` is the depth, so to print it to one decimal place, you could use:

```
print "%.1f" % AllOut[1][3]
```

From this first query, the script then uses the specimen information define a second query. This query is used to isolate a row of the CTD data that corresponds to the same dive and depth at which that specimen was collected. This approach lets you use information from one table to guide the extraction of information from another table.

Looking ahead

This is just the briefest view of data management with relational databases. There are many other ways to combine data across tables, and important best practices to follow to keep your database from growing unwieldy as it gets larger. If you would like to use databases in your research, we strongly suggest that you continue learning about them by following up this chapter with other resources, including the resources indicated at the end of the chapter.

Database users and security

All of the MySQL examples in this chapter have been executed as the `root` user on the `localhost` database system without password protection. This was done only to simplify the introduction to database systems. If you use databases in your research, and certainly if you are using a database with network access or on a shared computer, you should add password protection and create additional users with restricted privileges to access and modify the data. There are a couple of reasons for this. First, if you are logged in as `root` it is very simple to make a mistake that wipes out your entire database or makes widespread changes that you may not initially notice but that have a large effect on your data. If you log in as a user with restricted privileges, you create a line of defense against these mistakes. Second, if you provide network access to your database or it is installed on a shared machine, you will want to provide secure restricted access to the data. Even if someone isn't after your super-secret science data, it is an invitation to hackers to compromise your machine, whether to turn it to their own purposes or just because they enjoy wasting other people's time—as well as their own—with computer vandalism.

Creating a `root` password

SQL isn't only used for interacting with your data; it is also used to modify MySQL user and password settings. This is because the information about database system configuration and users is stored right in the database itself. The command for changing the password of an existing user is `SET PASSWORD`. If you don't specify a particular user when you issue the command, it changes the password of the user currently logged in. The following example changes the password for the root MySQL user to `mypass`, and then shows how to log in with the new password:

```
$ mysql -u root
mysql> SET PASSWORD = PASSWORD('mypass');
Query OK, 0 rows affected (0.83 sec)
mysql> EXIT;
Bye
$ mysql -u root -p
Enter password:
mysql>
```