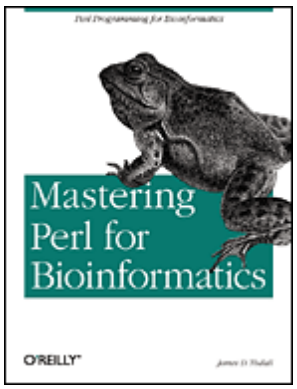
Mastering Perl for Bioinformatics

By James Tisdall

START READING

Publisher: O'Reilly
Pub Date: September 2003
ISBN: 0-596-00307-2
Pages: 396

Mastering Perl for Bioinformatics covers the core Perl language and many of its module extensions, presenting them in the context of biological data and problems of pressing interest to the biological community. This book, along with Beginning Perl for Bioinformatics, forms a basic course in Perl programming. This second volume finishes the basic Perl tutorial material (references, complex data structures, object-oriented programming, use of modules--all presented in a biological context) and presents some advanced topics of considerable interest in bioinformatics.



Mastering Perl for Bioinformatics

By James Tisdall

START READING

Publisher: O'Reilly
Pub Date: September 2003
ISBN: 0-596-00307-2
Pages: 396

Section B.6. Finding Help

Colophon
Index
[Team LiB]

PREVIOUS NEXT

Copyright

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Foreword

If you can't do bioinformatics, you can't do biology, and Perl is the biologist's favorite language for doing bioinformatics. The genomics revolution has so altered the landscape of biology that almost anyone who works at the bench now spends much of his time at the computer as well, browsing through the large online databases of genes, proteins, interactions and published papers. For example, the availability of an (almost) complete catalog of all the genes in human has fundamentally changed how anyone involved in genetic research works. Traditionally, a biologist would spend days thinking out the strategy for identifying a gene and months working in the lab cloning and screening to get his hands on it. Now he spends days thinking out the appropriate strategy for mining the gene from a genome database, seconds executing the query, and another few minutes ordering the appropriate clone from the resource center. The availability of genomes from many species and phyla makes it possible to apply comparative genomics techniques to the problems of identifying functionally significant portions of proteins or finding the genes responsible for a species' or strains distinguishing traits.

Parallel revolutions are occurring in neurobiology, in which new imaging techniques allow functional changes in the nervous systems of higher organisms to be observed in situ; in clinical research, where the computer database is rapidly replacing the paper chart; and even in botany, where herbaria are being digitized and cataloged for online access.

Biology is undergoing a sea change, evolving into an information-driven science in which the acquisition of large-scale data sets followed by pattern recognition and data mining plays just as prominent a role as traditional hypothesis testing. The two approaches are complementary: the patterns discovered in large-scale data sets suggest hypotheses to test, while hypotheses can be tested directly on the data sets stored in online databases.

To take advantage of the new biology, biologists must be as comfortable with the computer as they now are with thermocyclers and electrophoresis units. Web-based access to biological databases and the various collections of prepackaged data analysis tools are wonderful, but often they are not quite enough. To really make the most of the information revolution in biology, biologists must be able to manage and analyze large amounts of data obtained from many different sources. This means writing software. The ability to create a Perl script to automate information management is a great advantage: whether the task is as simple as checking a remote web page for updates or as complex as knitting together a large number of third-party software packages into an analytic pipeline.

In his first bioinformatics book, Beginning Perl for Bioinformatics, Jim introduced the fundamentals of programming in the language most widely used in the field. This book goes the next step, showing how Perl can be used to create large software projects that are scalable and reusable. If you are programming in Perl now and have experienced that wave of panic when you go back to some code you wrote six months ago and can't understand how the code works, then you know why you need this book. If you are an accomplished programmer who has heard about bioinformatics and wants to learn more, this book is also for you. Finally, if you are a biologist who wants to ride the crest of the information wave rather than being washed underneath it, then buy both this book along with Beginning Perl for Bioinformatics. I promise you won't be disappointed.

—Lincoln SteinCold Spring Harbor, NYSeptember 2003

Preface

The history of biological research is filled with examples of new laboratory techniques which, at first, are suitable topics for doctoral theses but eventually become so widely useful and standard that they are learned by most undergraduates. The use of computer programming in biology research is such an increasingly standard skill for many biologists. Bioinformatics is one of the most rapidly growing areas of biological science. Fundamentally, it's a cross-disciplinary study, combining the questions of computer science and programming with those of biological research.

As active sciences evolve, unifying principles and techniques developed in one field are often found to be useful in other areas. As a result, the established boundaries between disciplines are sometimes blurred, and the new principles and techniques may result in new ways of seeing the science as a whole. For instance, molecular biology has developed a set of techniques over the past 50 years that has also proved useful throughout much of biology in general. Similarly, the methods of bioinformatics are finding fertile ground in such fields as genetics, biochemistry, molecular biology, evolutionary science, development, cell studies, clinical research, and field biology.

In my view, *bioinformatics*, which I define broadly as the use of computers in biological research, is becoming a foundational science for a broad range of biological studies. Just as it's now commonplace to find a geneticist or a field biologist using the techniques of molecular biology as a routine part of her research, so can you frequently find that same researcher applying the techniques of bioinformatics. Molecular biology and bioinformatics may not be the researcher's main areas of interest, but the tools from molecular biology and bioinformatics have become standard in searching for the answers to the questions of interest. The Perl programming language plays no small part in that search for answers.

About This Book

This book is a continuation of my previous book, Beginning Perl for Bioinformatics (also by O'Reilly & Associates). As the title implies, Mastering Perl for Bioinformatics moves you to a more advanced level of Perl programming in bioinformatics. In this volume, I cover such topics as advanced data structures, object-oriented programming, modules, relational databases, web programming, and more advanced algorithms. The main goal of this book is to help you learn to write Perl programs that support your research in biology and enable you to adapt and use programs written by others.

In the process of honing your programming skills, you will also learn the fundamentals of bioinformatics. For many readers, the material presented in these two books will be sufficient to support their goals in the laboratory. However, this book is not a comprehensive survey of bioinformatics techniques. Both Mastering Perl for Bioinformatics and Beginning Perl for Bioinformatics emphasize the computer programming aspects of bioinformatics. As a serious student, you should expect to follow this groundwork with further study in the bioinformatics literature. Even the Perl programming language has more complexity than can fit in this cross-disciplinary text.

Readers already familiar with basic Perl and the elements of DNA and proteins can use Mastering Perl for Bioinformatics without reference to Beginning Perl for Bioinformatics. However, the two books together make a complete course suitable for undergraduates, graduate students, and professional biologists who need to learn programming for biology research.

A companion web site at http://www.oreilly.com/catalog/mperlbio includes all the program code in the book.

What You Need to Know to Use This Book

This book assumes that you have some experience with Perl, including a working knowledge of writing, saving, and running programs; basic Perl syntax; control structures such as loops and conditional tests; the most common operators such as addition, subtraction, and string concatenation; input and output from the user, files, and other programs; subroutines; the basic data types of scalar, array, and hash; and regular expressions for searching and for altering strings. In other words, you should be able to program Perl well enough to extract data from sources such as GenBank and the Protein Data Bank using pattern matching and regular expressions.

If you are new to Perl but feel you can forge ahead using a language summary and examples of programs, Appendix A provides a summary of the important parts of the Perl language. Previous programming experience in a high-level language such as C, Java, or FORTRAN (or any similar language); some experience at using subroutines to break a large problem into smaller, appropriately interrelated parts; and a tinkerer's delight in taking things apart and seeing what makes them tick may be all the computer-science prerequisites you need.

This book is primarily written for biologists, so it assumes you know the elementary facts about DNA, proteins, and restriction enzymes; how to represent DNA and protein data in a Perl program; how to search for motifs; and the structure and use of the databases GenBank, PDB, and Rebase. Because the book assumes you are a biologist, biology concepts are not explained in detail in order to concentrate on programming skills.

Biological data appears in many forms. The most important sources of biological data include the repository of public genetic data called GenBank (Genetic Data Bank) and the repository of public protein structure data called PDB (Protein Data Bank). Many other similar sources of biological data such as Rebase (Restriction Enzyme Database) are in wide use. All the databases just mentioned are most commonly distributed as text files, which makes Perl a good programming tool to find and extract information from the databases.

Organization of This Book

Here's a quick summary of what the book covers. If you're still relatively new to Perl you may want to work through the chapters in order. If you have some programming experience and are looking for ways to approach problems in bioinformatics with Perl, feel free to skip around.

Part I

Chapter 1

Modules are the standard Perl way of "packaging" useful programs so that other programmers can easily use previous work. Such standard modules as CGI, for instance, put the power of interactive web site programming within reach of a programmer who knows basic Perl. Also discussed in later chapters are Bioperl, for manipulating biological data, and DBI, for gaining access to relational databases. Modules are sometimes considered the most important part of Perl because that's where a lot of the functionality of Perl has been placed. In this chapter I show how to write your own modules, as well as how to find useful modules and use them in your programs.

Chapter 2

Complex data structures and references are fundamentally important to Perl. The basic Perl data structures of scalar, array, and hash go a long way toward solving many (perhaps most) Perl programming problems. However, many commonly used data structures such as multidimensional arrays, for instance, require more sophisticated Perl data structures to handle them. Perl enables you to define quite complex data structures, and we'll see how all that works.

String algorithms are standard techniques used in bioinformatics for finding important data in biological sequences; with them, you can compare two sequences, align two or more sequences, assemble a collection of sequence fragments, and so forth. String algorithms underlie many of the most commonly used programs in biology research, such as BLAST. In this chapter, a string matching algorithm that finds the closest match to a motif, based on the technique of dynamic programming, is presented in the form of a working Perl program.

Chapter 3

Object-oriented programming is a standard approach to designing programs. I assume, as a prerequisite, that you are familiar with the programming style called declarative programming. (For example, C and FORTRAN are declarative; C++ and Java are object-oriented; Perl can be either.) It's important for the Perl programmer to be familiar with the object-oriented approach. For instance, modules are usually defined in an object-oriented manner.

This chapter presents, step by step, the concepts and techniques of object-oriented Perl programming, in the context of a module that defines a simple class for keeping track of genes.

Chapter 4

In this chapter, object-oriented programming is further explored in the context of developing software to convert sequence files to alternate formats (FASTA, GCG, etc.). The concept of class inheritance is introduced and implemented.

Chapter 5

This chapter further develops object-oriented programming by writing a class that handles Rebase restriction enzyme data, a class that calculates restriction maps, and a class that draws restriction maps.

Part II

Chapter 6

Relational databases are important in programming because they save, organize, and retrieve data sets. This chapter introduces relational databases and the SQL language and includes information on designing and administering

Conventions Used in This Book

The following conventions are used in this book:

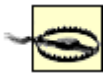
Constant width

Used for arrays, classes, code examples, loops, modules, namespaces, objects, packages, statements, and to show the output of commands.

Italics

Used for commands, directory names, filenames, example URLs, variables, and for new terms where they are defined.

This icon designates a note, which is an important aside to the nearby text.



This icon designates a warning relating to the nearby text.

Comments and Questions

Please address comments and questions concerning this book to the publisher:
O'Reilly & Associates, Inc. 1005 Gravenstein Highway NorthSebastopol, CA 95472(800) 998-9938 (in the United States or Canada)(707) 829-0515 (international or local)(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:
http://www.oreilly.com/catalog/mperlbio

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:
http://www.oreilly.com

Acknowledgments

My editor, Lorrie LeJeune, deserves special thanks for her work in developing the bioinformatics titles at O'Reilly. Her level of expertise is rare in any field. I thank Lorrie, Tim O'Reilly, and their colleagues for making it possible to bring these books to the public. I thank my technical reviewers for their invaluable expert help: Joel Greshock, Joe Johnston, Andrew Martin, and Sean Quinlan. I also thank Dr. Michael Caudy for his helpful suggestions in Chapter 3. I thank again those individuals mentioned in the first volume, especially those friends who have supported me during the writing of this book. I am also grateful to all those readers of the first volume who took the time and trouble to point out errors and weaknesses; their comments have substantially improved this volume as well. I thank Eamon Grennan and Jay Parini for their patient help with my writing. And I especially thank my much-loved children Rose, Eamon, and Joe, who are my most sincere teachers.

Part I: Object-Oriented Programming in Perl

Chapter 1. Modular Programming with Perl

Perl modules are essential to any Perl programmer. They are a great way to organize code into logical collections of interacting parts. They collect useful Perl subroutines and provide them to other programs (and programmers) in an organized and convenient fashion.

This chapter begins with a discussion of the reasons for organizing Perl code into modules. Modules are comparable to subroutines: both organize Perl code in convenient, reusable "chunks."

Later in this chapter, I'll introduce a small module, GeneticCode.pm. This example shows how to create simple modules, and I'll give examples of programs that use this module.

I'll also demonstrate how to find, install, and use modules taken from the all-important CPAN collection. A familiarity with searching and using CPAN is an essential skill for Perl programmers; it will help you avoid lots of unnecessary work. With CPAN, you can easily find and use code written by excellent programmers and road-tested by the Perl community. Using proven code and writing less of your own, you'll save time, money, and headaches.

1.1 What Is a Module?

A Perl module is a library file that uses package declarations to create its own namespace. Perl modules provide an extra level of protection from name collisions beyond that provided by my and use strict. They also serve as the basic mechanism for defining object-oriented classes.

1.2 Why Perl Modules?

Building a medium- to large-sized program usually requires you to divide tasks into several smaller, more manageable, and more interactive pieces. (A rule of thumb is that each "piece" should be about one or two printed pages in length, but this is just a general guideline.) An analogy can be made to building a microarray machine, which requires that you construct separate interacting pieces such as housing, temperature sensors and controls, robot arms to position the pipettes, hydraulic injection devices, and computer guidance for all these systems.

1.2.1 Subroutines and Software Engineering

Subroutines divide a large programming job into more manageable pieces. Modern programming languages all provide subroutines, which are also called functions, coroutines, or macros in other programming languages.

A subroutine lets you write a piece of code that performs some part of a desired computation (e.g., determining the length of DNA sequence). This code is written once and then can be called frequently throughout the main program. Using subroutines speeds the time it takes to write the main program, makes it more reliable by avoiding duplicated sections (which can get out of sync and make the program longer), and makes the entire program easier to test. A useful subroutine can be used by other programs as well, saving you development time in the future. As long as the inputs and outputs to the subroutine remain the same, its internal workings can be altered and improved without worrying about how the changes will affect the rest of the program. This is known as encapsulation.

The benefits of subroutines that I've just outlined also apply to other approaches in software engineering. Perl modules are a technique within a larger umbrella of techniques known as software encapsulation and reuse. Software encapsulation and reuse are fundamental to object-oriented programming.

A related design principle is *abstraction*, which involves writing code that is usable in many different situations. Let's say you write a subroutine that adds the fragment TTTTT to the end of a string of DNA. If you then want to add the fragment AAAAA to the end of a string of DNA, you have to write another subroutine. To avoid writing two subroutines, you can write one that's more abstract and adds to the end of a string of DNA whatever fragment you give it as an argument. Using the principle of abstraction, you've saved yourself half the work.

Here is an example of a Perl subroutine that takes two strings of DNA as inputs and returns the second one appended to the end of the first:

```
sub DNAappend {
        my ($dna, $tail) = @_;

        return($dna . $tail);
}
```

This subroutine can be used as follows:

```
my $dna = 'ACCGGAGTTGACTCTCCGAATA';
my $polyT = 'TTTTTTTT';

print DNAappend($dna, $polyT);
```

If you wish, you can also define subroutines polyT and polyA like so:

```
sub polyT {
    my ($dna) = @_;

    return DNAappend($dna, 'TTTTTTTT');
```

1.3 Namespaces

A *namespace* is implemented as a table containing the names of the variables and subroutines in a program. The table itself is called a *symbol table* and is used by the running program to keep track of variable values and subroutine definitions as the program evolves. A namespace and a symbol table are essentially the same thing. A namespace exists under the hood for many programs, especially those in which only one default namespace is used.

Large programs often accidentally use the same variable name for different variables in different parts of the program. These identically named variables may unintentionally interact with each other and cause serious, hard-to-find errors. This situation is called namespace collision. Separate namespaces are one way to avoid namespace collision.

The package declaration described in the next section is one way to assign separate namespaces to different parts of your code. It gives strong protection against accidentally using a variable name that's used in another part of the program and having the two identically-named variables interact in unwanted ways.

1.3.1 Namespaces Compared with Scoping: my and use strict

The unintentional interaction between variables with the same name is enough of a problem that Perl provides more than one way to avoid it. You are probably already familiar with the use of my to restrict the scope of a variable to its enclosing block (between matching curly braces {}) and should be accustomed to using the directive use strict to require the use of my for all variables. use strict and my are a great way to protect your program from unintentional reuse of variable names. Make a habit of using my and working under use strict.

1.4 Packages

Packages are a different way to protect a program's variables from interacting unintentionally. In Perl, you can easily assign separate namespaces to entire sections of your code, which helps prevent namespace collisions and lets you create modules.

Packages are very easy to use. A one-line package declaration puts a new namespace in effect. Here's a simple example:

```
$dna = 'AAAAAAAAAA';
package Mouse;
$dna = 'CCCCCCCCCC';
package Celegans;
$dna = 'GGGGGGGGGG';
```

In this snippet, there are three variables, each with the same name, $dna. However, they are in three different packages, so they appear in three different symbol tables and are managed separately by the running Perl program.

The first line of the code is an assignment of a poly-A DNA fragment to a variable $dna. Because no package is explicitly named, this $dna variable appears in the default namespace main.

The second line of code introduces a new namespace for variable and subroutine definitions by declaring package Mouse;. At this point, the main namespace is no longer active, and the Mouse namespace is brought into play. Note that the name of the namespace is capitalized; it's a well-established convention you should follow. The only noncapitalized namespace you should use is the default main.

Now that the Mouse namespace is in effect, the third line of code, which declares a variable, $dna, is actually declaring a separate variable unrelated to the first. It contains a poly-C fragment of DNA.

Finally, the last two lines of code declare a new package called Celegans and a new variable, also called $dna, that stores a poly-G DNA fragment.

To use these three $dna variables, you need to explicitly state which packages you want the variables from, as the following code fragment demonstrates:

```
print "The DNA from the main package:\n\n";
print $main::dna, "\n\n";

print "The DNA from the Mouse package:\n\n";
print $Mouse::dna, "\n\n";

print "The DNA from the Celegans package:\n\n";
print $Celegans::dna, "\n\n";
```

This gives the following output:

```
The DNA from the main package:

AAAAAAAAAA

The DNA from the Mouse package:

CCCCCCCCCC
```

1.5 Defining Modules

To begin, take a file of subroutine definitions and call it something like Newmodule.pm. Now, edit the file and give it a new first line:

```
package Newmodule;
```

and a new last line 1;. You've now created a Perl module.

To make a Celegans module, place subroutines in a file called Celegans.pm, and add a first line:

```
package Celegans;
```

Add a last line 1;, and you've defined a Celegans module. This last line just ensures that the library returns a true value when it's read in. It's annoying, but necessary.

1.6 Storing Modules

Where you store your .pm module files on your computer affects the name of the module, so let's take a moment to sort out the most important points. For all the details, consult the perlmod and the perlmodlib parts of the Perl documentation at http://www.perldoc.org. You can also type perldoc perlmod or perldoc perlmodlib at a shell prompt or in a command window.

Once you start using multiple files for your program code, which happens if you're defining and using modules, Perl needs to be able to find these various files; it provides a few different ways to do so.

The simplest method is to put all your program files, including your modules, in the same directory and run your programs from that directory. Here's how the module file Celegans.pm is loaded from another program:

```
use Celegans;
```

However, it's often not so simple. Perl uses modules extensively; many are built-in when you install Perl, and many more are available from CPAN, as you'll see later. Some modules are used frequently, some rarely; many modules call other modules, which in turn call still other modules.

To organize the many modules a Perl program might need, you should place them in certain standard directories or in your own development directories. Perl needs to know where these directories are so that when a module is called in a program, it can search the directories, find the file that contains the module, and load it in.

When Perl was installed on your computer, a list of directories in which to find modules was configured. Every time a Perl program on your computer refers to a module, Perl looks in those directories. To see those directories, you only need to run a Perl program and examine the built-in array @INC, like so:

```
print join("\n", @INC), "\n";
```

On my Linux computer, I get the following output from that statement:

```
/usr/local/lib/perl5/5.8.0/i686-linux
/usr/local/lib/perl5/5.8.0
/usr/local/lib/perl5/site_perl/5.8.0/i686-linux
/usr/local/lib/perl5/site_perl/5.8.0
/usr/local/lib/perl5/site_perl/5.6.1
/usr/local/lib/perl5/site_perl/5.6.0
/usr/local/lib/perl5/site_perl
.
```

These are all locations in which the standard Perl modules live on my Linux computer. @INC is simply an array whose entries are directories on your computer. The way it looks depends on how your computer is configured and your operating system (for instance, Unix computers handle directories a bit differently than Windows).

Note that the last line of that list of directories is a solitary period. This is shorthand for "the current directory," that is, whatever directory you happen to be in when you run your Perl program. If this directory is on the list, and you run your program from that directory as well, Perl will find the .pm files.

When you develop Perl software that uses modules, you should put all the modules together in a certain directory. In order for Perl to find this directory, and load the modules, you need to add a line before the use MODULE directives, telling Perl to additionally search your own module directory for any modules requested in your program. For instance, if I put a module I'm developing for my program into a file named Celegans.pm, and put the

1.7 Writing Your First Perl Module

Now that you've been introduced to the basic ideas of modules, it's time to actually examine a working example of a module.

In this section, we'll write a module called Geneticcode.pm, which implements the genetic code that maps DNA codons to amino acids and then translates a string of DNA sequence data to a protein fragment.

1.7.1 An Example: Geneticcode.pm

Let's start by creating a file called Geneticcode.pm and using it to define the mapping of codons to amino acids in a hash variable called %genetic_code. We'll also discuss a subroutine called codon2aa that uses the hash to translate its codon arguments into amino acid return values.

Here are the contents of the first module file Geneticcode.pm:

```
package Geneticcode;

use strict;
use warnings;

my(%genetic_code) = (

    'TCA' => 'S',    # Serine
    'TCC' => 'S',    # Serine
    'TCG' => 'S',    # Serine
    'TCT' => 'S',    # Serine
    'TTC' => 'F',    # Phenylalanine
    'TTT' => 'F',    # Phenylalanine
    'TTA' => 'L',    # Leucine
    'TTG' => 'L',    # Leucine
    'TAC' => 'Y',    # Tyrosine
    'TAT' => 'Y',    # Tyrosine
    'TAA' => '_',    # Stop
    'TAG' => '_',    # Stop
    'TGC' => 'C',    # Cysteine
    'TGT' => 'C',    # Cysteine
    'TGA' => '_',    # Stop
    'TGG' => 'W',    # Tryptophan
    'CTA' => 'L',    # Leucine
    'CTC' => 'L',    # Leucine
    'CTG' => 'L',    # Leucine
    'CTT' => 'L',    # Leucine
    'CCA' => 'P',    # Proline
    'CCC' => 'P',    # Proline
    'CCG' => 'P',    # Proline
    'CCT' => 'P',    # Proline
    'CAC' => 'H',    # Histidine
    'CAT' => 'H',    # Histidine
    'CAA' => 'Q',    # Glutamine
    'CAG' => 'Q',    # Glutamine
    'CGA' => 'R',    # Arginine
    'CGC' => 'R',    # Arginine
    'CGG' => 'R',    # Arginine
    'CGT' => 'R',    # Arginine
    'ATA' => 'I',    # Isoleucine
    'ATC' => 'I',    # Isoleucine
    'ATT' => 'I',    # Isoleucine
    'ATG' => 'M',    # Methionine
```

1.8 Using Modules

So far, the benefit of modules may seem questionable. You may be wondering what the advantage is over simple libraries (without package declarations), since the main result seems to be the necessity to refer to subroutines in the modules with longer names!

1.8.1 Exporting Names

There's a way to avoid lengthy module names and still use the short ones if you place a call to the special Exporter module in the module code and modify the use MODULE declaration in the calling code.

Going back to the first example Geneticcode.pm module, recall it began with this line:

```
package Geneticcode;
```

and included the definition for the hash genetic_code and the subroutine codon2aa.

If you add these lines to the beginning of the file, you can export the symbol names of variables or subroutines in the module into the namespace of the calling program. You can then use the convenient short names for things (e.g., codon2aa instead of Geneticcode::codon2aa). Here's a short example of how it works (try typing perldoc Exporter to see the whole story):

```
package Geneticcode;

require Exporter;
@ISA = qw(Exporter);

@EXPORT_OK = qw(...);          # symbols to export on request
```

Here's how to export the name codon2aa from the module only when explicitly requested:

```
@EXPORT_OK = qw(codon2aa);     # symbols to export on request
```

The calling program then has to explicitly request the codon2aa symbol like so:

```
use Geneticcode qw(codon2aa);
```

If you use this approach, the calling program can just say:

```
codon2aa($codon);
```

instead of:

```
Geneticcode::codon2aa($codon);
```

The Exporter module that's included in the standard Perl distribution has several other optional behaviors, but the way just shown is the safest and most useful. As you'll see, the object-oriented programming style of using modules doesn't use the Export facility, but it is a useful thing to have in your bag of tricks. For more information about exporting (such as why exporting is also known as "polluting your namespace"), see the Perl documentation for the Exporter module (by typing perldoc Exporter at a command line or by going to the http://www.perldoc.com web page).

1.9 CPAN Modules

The Comprehensive Perl Archive Network (CPAN, http://www.cpan.org) is an impressively large collection of Perl code (mostly Perl modules). CPAN is easily accessible and searchable on the Web, and you can use its modules for a variety of programming tasks.

By now you should have the basic idea of how modules are defined and used, so let's take some time to explore CPAN to see what goodies are available.

There are two important points about CPAN. First, a large number of the things you might want your programs to do have already been programmed and are easily obtained in downloadable modules. You just have to go find them at CPAN, install them on your computer, and call them from your program. We'll take a look at an example of exactly that in this section.

Second, all code on CPAN is free of charge and available for use by a very unrestrictive copyright declaration. Sound good? Keep reading.

CPAN includes convenient ways to search for useful modules, and there's a CPAN.pm module built-in with Perl that makes downloading and installing modules quite easy (when things work well, which they usually do). If you can't find CPAN.pm, you should consider updating your current version.

You can find more information by typing the following at the command line:

```
perldoc CPAN
```

You can also check the Frequently Asked Questions (FAQ) available at the CPAN web site.

1.9.1 What's Available at CPAN?

The CPAN web site offers several "views" of the CPAN collection of modules and several alternate ways of searching (by module name, category, full text search of the module documentation, etc.). Here is the top-level organization of the modules by overall category:

```
Development Support
Operating System Interfaces
Networking Devices IPC
Data Type Utilities
Database Interfaces
User Interfaces
Language Interfaces
File Names Systems Locking
String Lang Text Proc
Opt Arg Param Proc
Internationalization Locale
Security and Encryption
World Wide Web HTML HTTP CGI
Server and Daemon Utilities
Archiving and Compression
Images Pixmaps Bitmaps
Mail and Usenet News
Control Flow Utilities
File Handle Input Output
Microsoft Windows Modules
```

1.10 Exercises

Exercise 1.1

What are the problems that might arise when dividing program code into separate module files?

Exercise 1.2

What are the differences between libraries, modules, packages, and namespaces?

Exercise 1.3

Write a module that finds modules on your computer.

Exercise 1.4

Where do the standard Perl distribution modules live on your computer?

Exercise 1.5

Research how Perl manages its namespaces.

Exercise 1.6

When might it be necessary to export names from a module? When might it be useful? When might it be convenient? When might it be a very bad idea?

Exercise 1.7

The program testGeneticcode contains the following loop:

```
 # Translate each three-base codon to an amino acid, and append to a protein
for(my $i=0; $i < (length($dna) - 2) ; $i += 3) {
        $protein .= Geneticcode::codon2aa( substr($dna,$i,3) );
}
```

Here's another way to accomplish that loop:

```
 # Translate each three-base codon to an amino acid, and append to a protein
my $i=0;
while (my $codon = substr($dna, $i += 3, 3) ) {
        $protein .= Geneticcode::codon2aa( $codon );
}
```

Compare the two methods. Which is easier to understand? Which is easier to maintain? Which is faster? Why?

Exercise 1.8

The subroutine codon2aa causes the entire program to halt when it encounters a "bad" codon in the data. Often (usually) it is best for a subroutine to return some indication that it encountered a problem and let the calling program decide how to handle it. It makes the subroutine more generally useful if it isn't always halting the program (although that is what you want to do sometimes).

Rewrite codon2aa and the calling program testGeneticcode so that the subroutine returns some error—perhaps the value undef—and the calling program checks for that error and performs some action.

Exercise 1.9

Write a separate module for each of the following: reading a file, extracting FASTA sequence data, and printing sequence data to the screen.

Exercise 1.10

Download, install, and use a module from CPAN.

Chapter 2. Data Structures and String Algorithms

So far in this book, I've used the standard Perl data structures of scalars, arrays, and hashes. However, it is often necessary to handle data with a more complex structure than what those basics allow. For instance, it is frequently useful to have a two-dimensional array.

In this chapter, you'll learn how to define and use references and complex data structures. After you learn the fundamentals, you'll apply the new techniques to implement a biologically important algorithm. These techniques are also fundamental to the implementation of object-oriented programming, as you'll see in Chapter 3.

The algorithm we'll study is called approximate string matching. It lets you find the closest match for a peptide fragment in a protein, for instance. It uses an algorithmic technique called dynamic programming, an essential tool for many similar biological tasks, such as aligning biological sequences. In this chapter, you'll see how Perl references can be used to write programs for data problems with more complex relationships. References are also used for the objects of object-oriented programming.

2.1 Basic Perl Data Types

Before tackling references, let's review the basic Perl data types:
Scalar

A scalar value is a string or any one of several kinds of numbers such as integers, floating-point (decimal) numbers, or numbers in scientific notation such as 2.3E23. A scalar variable begins with the dollar sign $, as in $dna.
Array

An array is an ordered collection of scalar values. An array variable begins with an at sign @, as in @peptides. An array can be initialized by a list such as @peptides = ('zeroth', 'first', 'second'). Individual scalar elements of an array are referred to by first preceding the array name with a dollar sign (an individual element of an array is a scalar value) and then following the array name with the position of the desired element in square brackets. Thus the first element of the @peptides array is referenced by $peptides[0] and has the value 'zeroth'. (Note that array elements are given the positions 0, 1, 2, ..., n-1, where n is the number of elements in the array.)

Recall that printing an array within double quotes causes the elements to be separated by spaces; without the double quotes, the elements are printed one after the other without separations. This snippet:

```
@pentamers = ('cggca', 'tgatc', 'ttggc');

print "@pentamers", "\n";
print @pentamers, "\n";
```

produces the output:

```
cggca tgatc ttggc
cggcatgatcttggc
```

Hash

A hash is an unordered collection of key value pairs of scalar values. Each scalar key is associated with a scalar value. A hash variable begins with the percent sign %, as in %geneticmarkers. A hash can be initialized like an array, except that each pair of scalars are taken as a key with its value, as in:

The => symbol is just a synonym for a comma that makes it easier to see the key/value pairs in such lists.[1] An individual scalar value is retrieved by preceding the hash name with a dollar sign (an individual value is a scalar value) and following the hash name with the key in curly braces, as in $geneticmarkers{'hairless'}, which, because of how it's initialized, has the value 'no'.

[1] It also forces the left side to be interpreted as a string.

2.2 References

Many computer languages provide variables that allow you to refer to, or point at, other values. So, instead of a variable containing data such as a string or number of interest, the variable contains the location of the data; it tells you where to go to get the value you want. In Perl, the use of a scalar variable to refer to another value is called a reference, and the value being pointed at is called a referent.

References allow you to do many useful things in Perl; you can define multidimensional arrays and other more complex data structures and avoid copying large amounts of data (for instance, when passing arguments into subroutines). Using references can make your programs faster, more efficient, and shorter. References have a number of uses, as you'll see in the next sections.

2.2.1 References to Scalars

Here's an example of a reference:

```
$peptide = 'EIQADEVRL';

$peptideref = \$peptide;

print "Here is what's in the reference:\n";
print $peptideref, "\n";

print "Here is what the reference is pointing to:\n";
print ${$peptideref}, "\n";
print $$peptideref, "\n";
```

This Perl code produces the following output:

```
Here is what's in the reference:
SCALAR(0x80fe4ac)
Here is what the reference is pointing to:
EIQADEVRL
EIQADEVRL
```

What's going on here?

First, a string value of EIQADEVRL is assigned to the scalar variable $peptide. Next, a backslash operator is used before the $peptide variable to return a reference to the variable. This reference is saved in the scalar variable $peptideref.

The next lines of code show what this example really does. When you print out the (actual) value of the reference variable $peptideref, you get the value:

```
SCALAR(0x80fe4ac)
```

This says that the reference variable $peptideref is pointing to a scalar value (which is the value of the scalar variable $peptide). It also gives a hexadecimal number that specifies where in the computer memory the value for that variable resides.

The 0x at the beginning of the number says that it is a hexadecimal number.[2] Hexadecimal (base 16) numbers are a way to specify locations in computer memory. The exact location in the computer memory where this $peptide value resides is almost never of practical importance to you. However, it can help when debugging code that uses references, and so it is displayed when you print the value of a reference as we've just done or when you use the Perl

2.3 Matrices

Perl matrices are built from simpler data structures using references. Recall that a matrix is a set of values that can be uniquely referenced by indexes. If only one index is required, the matrix is one-dimensional (this is exactly how an array works in Perl). If n indexes are required, the matrix is n-dimensional.

2.3.1 Two-Dimensional Matrices

A two-dimensional matrix is one of the simplest complex data structures. It can be conceptualized as a table of rows and columns, in which each element of the table is uniquely identified by its particular row and column.

There are several ways to build matrices in Perl. We'll look at some of the most useful.

Because there is no built-in matrix data structure, you have to build a matrix from other data structures. The most straightforward way to do this is with an array of arrays:

```
@probes = (
    [1, 3, 2, 9],
    [2, 0, 8, 1],
    [5, 4, 6, 7],
    [1, 9, 2, 8]
);

print "The probe at row 1, column 2 has value ", $probes[1][2], "\n";
```

This prints out:

```
The probe at row 1, column 2 has value 8
```

Recall that in Perl the first element of an array is indexed 0; so row 1 in this program is actually the second row, and column 2 is actually the third column. Sometimes you may want to refer to the 0th row as row 1; you have to adjust your code and your interactions with the user accordingly.

This matrix is implemented as an array (in parentheses), each element of which is a reference to an anonymous array [in square brackets], which itself is a list of integers.

Another good way to build an array is to declare a reference to an anonymous array. In the following example, I declare an empty anonymous array and then populate it as desired. This is, in effect, an anonymous array of anonymous arrays:

```
# Declare reference to (empty) anonymous array
$array = [  ];

# Initialize the array
for($i=0; $i < 4 ; ++$i) {
  for($j=0; $j < 4 ; ++$j) {
      $array->[$i][$j] = $i * $j;
  }
}

# Reset one of the elements of the array
$array->[3][2] = 99;
```

2.4 Complex Data Structures

Different algorithms require different data structures. Using references in Perl, it is possible to build very complex data structures.

This section gives a short introduction to some of the possibilities, such as a hash with array values and a two-dimensional array of hashes. See the recommended reading in Section 2.9 of this chapter for books and sections of the Perl manual that are very helpful.

Perl uses the basic data types of scalar, array, and hash, plus the ability to declare scalar references to those basic data types, to build more complex structures. For instance, an array must have scalar elements, but those scalar elements can be references to hashes, in which case you have effectively created an array of hashes.

2.4.1 Hash with Array Values

A common example of a complex data structure is a hash with array values. Using such a data structure, you can associate a list of items with each keyword. The following code shows an example of how to build and manage such a data structure. Assume you have a set of human genes, and for each human gene, you want to manage an array of organisms that are known to have closely related genes. Of course, each such array of related organisms can be a different length:

```
use Data::Dumper;

%relatedgenes = (  );

$relatedgenes{'stromelysin'} = [
    'C.elegans',
    'Arabidopsis thaliana'
];
$relatedgenes{'obesity'} = [
    'Drosophila',
    'Mus musculus'
];

# Now add a new related organism to the entry for 'stromelysin'

push( @{$relatedgenes{'stromelysin'}}, 'Canis' );

print Dumper(\%relatedgenes);
```

This program prints out the following (the very useful Data::Dumper module is described in more detail later; try typing perldoc Data::Dumper for the details of this useful way to print out complex data structures):

```
$VAR1 = {
        'stromelysin' => [
                           'C.elegans',
                           'Arabidopsis thaliana',
                           'Canis'
                         ],
        'obesity' => [
                       'Drosophila',
                       'Mus musculus'
                     ]
};
```

The tricky part of this short program is the push. The first argument to push must be an array. In the program, this array is @{$relatedgenes{'stromelysin'}}. Examining this array from the inside out, you can see that it refers to the

2.5 Printing Complex Data Structures

Sometimes you need to look inside your complex data structures to see what the settings are. One of the most useful ways to examine a data structure is by means of the Data::Dumper module. This module comes standard with all recent versions of Perl.

Here is the summary and part of the synopsis and description as output from the perldoc Data::Dumper command:

```
NAME
       Data::Dumper - stringified perl data structures, suitable
       for both printing and "eval"

SYNOPSIS
           use Data::Dumper;

           # simple procedural interface
           print Dumper($foo, $bar);
(...)

DESCRIPTION
       Given a list of scalars or reference variables, writes out
       their contents in perl syntax. The references can also be
       objects.  The contents of each variable is output in a
       single Perl statement.  Handles self-referential strucTures correctly.

       The return value can be "eval"ed to get back an identical
       copy of the original reference structure.
(...)
```

This output of a two-dimensional array illustrates its use:

```
use Data::Dumper;

$array = [  ];

# Initialize the array
for($i=0; $i < 4 ; ++$i) {
  for($j=0; $j < 4 ; ++$j) {
      $array->[$i][$j] = $i * $j;
  }
}

# Print the array "by hand"
for($i=0; $i < 4 ; ++$i) {
  for($j=0; $j < 4 ; ++$j) {
      printf("%3d ", $array->[$i][$j]);
  }
  print "\n";
}

# Print the array using Data::Dumper
print Dumper($array);
```

This produces the output:

```
   0   0   0   0
  0   1   2   3
  0   2   4   6
  0   3   6   9
$VAR1 = [
          [
            0,
            0,
```

2.6 Data Structures in Action

The previous sections introduced a fair amount of new Perl syntax and capabilities. Now, let's see some of these new capabilities in action.

2.6.1 The Problem of String Matching

It is frequently important in biology to find the best possible match for a short sequence in a longer sequence; for example, between an oligonucleotide and the sequence of DNA that has been cloned in a YAC or BAC. This match need not always be perfect; frequently, what is important is to find the closest match available. This problem is known in computer science as approximate string matching, and dynamic programming is a popular technique used to compute the solution.

The problem of string matching is to find a pattern, such as a nucleotide or peptide fragment, in a longer text such as a chromosome or protein.

The problem of approximate string matching is to find a pattern in a text in which the match might not be perfect. Perhaps a few of the characters are different or missing; the problem is to find the best match possible.

2.6.2 Genetic Variability and String Matching

Biologically, approximate matches are of commanding importance. Evolutionary changes between species can make genes with essentially the same function collect a fair number of individual base changes; they may even have acquired differences in exon structure. Even within a species, individual base changes among groups in the population (single nucleotide polymorphisms) are important causes of disease and important clues in the discovery of disease-causing genes.

Mutations tend to accumulate over time in noncoding regions of DNA; mutations in coding regions tend to avoid altering critical regions essential for the functioning of the gene (where mutations may be fatal to the organism). Even a noncoding region may be critical to the regulation of a gene and thus tend to resist mutations. As a result, studying where mutations are not accumulating is often an important clue to discerning the function and control of a gene and its associated protein.

Due to the redundancy of the genetic code, mutations in DNA may not affect the coding of the associated protein. Other mutations may make a change in a coded amino acid, but it may be an amino acid with similar properties to the original amino acid; for instance, both amino acids may be hydrophilic. Tracing these kinds of mutations is another source of important information about the process of mutation. It can give vital clues to the conservation of critical coding regions and to the folding of proteins.

Biologists will have no difficulty expanding upon the preceding brief motivation for studying approximate string matching and other techniques that find similarities between biological molecules. Many standard laboratory techniques rely on the annealing of a molecule to another molecule with similar, but not necessarily exact, structure.

In the discussion that follows, you'll use your new knowledge of complex data structures in Perl to implement an algorithm that finds approximate matches of patterns in text, such as DNA fragments in chromosomes or peptide fragments in proteins.

2.7 Dynamic Programming

Dynamic programming computes the values for small subproblems and stores those values in a matrix. The stored values are then used to solve larger subproblems (without incurring the cost of recomputing the smaller subproblems) and so on until the solution to the overall problem is found. The term "dynamic programming" is a bit of a misnomer since it doesn't involve changing values over time as the word "dynamic" suggests.

This approach relies on having a data structure available to store the intermediate values as the algorithm proceeds. The data structure may require a fair amount of computer memory, but the overall speed of the algorithm often makes the memory cost worthwhile. In this section, we'll use a Perl multidimensional array, namely a simple two-dimensional matrix, to solve an approximate string matching problem.

Our algorithm will find a (shorter) pattern in a (longer) text. We'll start with a two-dimensional array, or matrix. The columns of the matrix will be associated with the (shorter) pattern, and the rows of the matrix will be associated with the (longer) text. The zeroth row and the zeroth column will be initialized to the appropriate starting values. We'll then calculate each value in the matrix by examining adjacent, already calculated values in conjunction with the characters of the pattern and the text. After the entire matrix has been filled in, we'll have the answer to our problem. That is, we'll find the position(s) in the text that most closely match the pattern, and we'll do so by simply examining the values in the last row of the matrix.

2.8 Approximate String Matching

You've most likely learned how to use regular expressions to find any of a set of possible patterns in a string. Approximate string matching is similar: an approximate string matching algorithm finds any of a set of possible patterns in a string. However, the two approaches are quite different in their capabilities and their ease of use. Simply stated, approximate string matching can find many close matches to a pattern that would be very tedious to specify using regular expressions.

2.8.1 Edit Distance

There are several ways to measure the distance between two strings, and our algorithm will use one such measure. Some variants of this measure are considered in the exercises at the end of the chapter.

Our algorithm uses the idea of edit distance to measure the similarity between two strings. The idea is quite simple. Assume that there are three things you can do to alter a string:
Substitution

Change any character to a different character
Deletion

Delete any character
Insertion

Insert a new character at any position

Now, let's say that every time you make any of these three edits, you incur an edit cost of 1. Now, call the edit distance between two strings as the minimum edit cost needed to change one string into the other.

For instance, let's say there are two strings portend and profound. You can apply the following edits to portend:

```
portend
        (delete o)
prtend
        (insert o)
protend
        (change t to f)
profend
        (change e to o)
profond
        (insert u)
profound
```

You can see that five edits were applied. Assuming you can't find a quicker way to change one string into the other, the edit distance between the two strings is 5.

Clearly, you can also start from the other string and apply the same edits in reverse (just interchanging the deletions and insertions, and reversing the substitutions). So, starting from profound, delete u, change o to e, change f to t, delete o, and finally insert o, to arrive at portend.

The relevance of this concept of edit distance to sequence alignment is simply: the smaller the edit distance, the better

2.9 Resources

I recommend the following O'Reilly sources for more details on data structures in Perl:

- Programming Perl, by Larry Wall, Tom Christiansen, and Jon Orwant. This is the bible of Perl programming. Everything about data structures is explained in detail.
- Advanced Perl Programming by Sriram Srinivasan. An excellent book that covers references and data structures.
- Mastering Algorithms with Perl by Jon Orwant, Jarkko Hietaniemi, and John Macdonald. A marvelous book, especially if you like this chapter. Many interesting data structures and algorithms are explained and implemented in Perl.
- The Perl Cookbook by Tom Christiansen and Nathan Torkington. As the title implies, this book is composed of fairly short recipes that accomplish particular tasks, grouped according to application area.

Here's where to go for Perl documentation:

- The perlreftut tutorial page from the Perl documentation gives a short introduction to Perl references (type perldoc perlreftut at your command line if Perl is installed, or visit the web page http://www.perldoc.com).
- The perlref tutorial page from the Perl documentation discusses Perl references in detail.
- The perldata tutorial page from the Perl documentation gives an introduction to Perl data structures.
- The perldsc tutorial page from the Perl documentation presents a "cookbook" overview of Perl data structures.
- The perllol tutorial page from the Perl documentation gives an introduction to arrays of arrays.

The literature on algorithms is vast, including many textbooks as well as advanced monographs and peer reviewed journals. I recommend the following sources as a few entry points for more details on dynamic programming and algorithms:

- Computer Algorithms by Sara Baase and Allen VanGelder (Addison Wesley). This book is clearly written for the undergraduate level, and includes a very nice explanation of the algorithm presented in this chapter.
-

2.10 Exercises

Exercise 2.1

Suggest a programming situation in which it would make sense to have several scalar references to one scalar variable that contains a peptide fragment.

Exercise 2.2

When might you want to use a reference to an (anonymous) scalar constant?

Exercise 2.3

Is $$arr[0] the same as $arr->[0]? Why or why not?

Exercise 2.4

Write a subroutine that returns a reference to a hash. Declare a reference to this subroutine and call it using the reference, then print out the hash whose reference is returned from the subroutine.

Exercise 2.5

Write a subroutine that returns a new anonymous subroutine based on its arguments, which are passed to it as references. Call the subroutine and then call the new subroutine that is returned.

Exercise 2.6

Write a subroutine to multiply two matrices.

Exercise 2.7

Develop a data structure that is a hash at the top level and can be used to record the data from microarray runs.

Exercise 2.8

Write a min subroutine that returns the minimum of two integers. Rewrite min3 using it.

Exercise 2.9

Make a subroutine that prints the distance matrix. Make it handle the display of longer numbers appropriately.

Exercise 2.10

Make the subroutine from Exercise 2.9 also display the pattern and string aligned with the output of the edit distance matrix.

Exercise 2.11

Make a subroutine that returns the edit distance array, the best score, and the locations of the best matches.

Exercise 2.12

Any difference in Exercise 2.11 if you calculate row by row instead of column by column?

Exercise 2.13

In Exercise 2.11, save space by keeping only two rows in memory.

Exercise 2.14

In Exercise 2.11, report on types of edits of matches.

Exercise 2.15

In Exercise 2.14, show the strings aligned, extra spaces for insertions or deletions, and flag mismatches.

Exercise 2.16

In Exercise 2.11, can you speed the program up by skipping computations when it becomes clear that the best score you've already found can't be matched or bettered in that column? Why or why not?

Exercise 2.17

Chapter 3. Object-Oriented Programming in Perl

In Chapter 1, you saw how modules are defined and used, and in Chapter 2, how references and data structures work. Now, it's time to introduce the important concepts and techniques of object-oriented programming in Perl that are based on modules and references.

Object-oriented (OO) programming is one of the most important approaches to writing programs, and it is an approach that has been well supported by Perl for quite a while. Other OO languages of interest include Java, C++, and Smalltalk. Many Perl modules are written in an OO style, and their proper use requires some fundamental understanding of the OO approach. Luckily, the key concepts are fairly simple.

Perl easily supports both declarative and OO programming. (Perl was originally a declarative language only; the OO style was added fairly early on.) Declarative programming is characterized by code that declares variables and subroutines, conditional tests, if-else branches, and loops, and various arithmetic, logical, and string operators. It is up to you to manage the definition and use of the variables and subroutines so that they interact in appropriate ways. (You'll see shortly how object-oriented programming imposes additional constraints that help you create well-behaved programs.) Many declarative programming languages are well established, including Perl and such stalwarts as C, FORTRAN, and BASIC, to name just a few. By this point, assuming you have some experience programming in Perl, you should be fairly comfortable with the declarative style.

The first part of this chapter is an overview of OO programming and how OO Perl modules are used. If you're a beginning Perl programmer, you'll find them easy to use because they rarely require you to know how to write OO Perl code. Depending on your needs and goals, this might be all the information you'll require from this chapter.

As a more advanced programmer, you'll sometimes need to write your own OO bioinformatics software. If you're such a programmer, the second part of this chapter will be of greatest interest to you. However, because the material is developed incrementally, you will most likely want to read the chapter in order from beginning to end.

Perl makes clever and simple use of existing mechanisms to support OO programming. Perl packages and modules are used to define OO classes, Perl references define OO objects, and Perl subroutines define OO methods. The definitions of these terms will become clear as you read the chapter, but in brief, OO software is organized into classes that contain data called objects. Subroutines called methods operate on the objects.

Over the course of this chapter, I'll develop a small example object module, Gene.pm, to demonstrate the essentials of OO Perl. Gene.pm is developed in four stages so you can learn the OO style gradually. The final code for Gene.pm serves as a template from which you can begin developing your own OO software.

3.1 What Is Object-Oriented Programming?

Object-oriented programming is a way to organize code so it interacts in certain prescribed ways, obeying certain rules about how the data and subroutines are organized. In other words, it imposes a certain programming discipline that can lead to better and more reliable code.

The key idea of OO programming is that all data is stored and modified with special data structures called objects, and each kind of object can be accessed only by its defined subroutines called methods. The user of an OO class is typically spared the effort of directly manipulating data, and can use class methods for this instead.

The promise of this OO structure of program code is that it makes the resulting programs cleanly designed, more reliable, easier to reuse in other programs, and easier to modify and improve. In essence, the approach imposes certain restrictions on what a programmer can do with the data and subroutines at hand.

Proponents of the OO approach cite the benefits this extra discipline provides. It is certainly true that you can follow good programming practices without using an OO approach. However, OO does provide a well-defined framework for encouraging discipline and good programming practices. In a very flexible language such as Perl, good practices can sometimes be easier to enforce in the framework of OO. We'll see how this comes about in the examples that follow.

3.1.1 Why Object-Oriented Programming?

It is often important and necessary to weigh the costs and benefits of a given system against the alternatives in an applied engineering discipline such as programming. The decision to use OO programming, declarative programming, or some other paradigm, is often subject to religious debates, with some enthusiasts promoting their favorite approach against all comers. This is especially relevant to the Perl programmer, because Perl allows you to write in the declarative or in the OO style. You should know that OO programming isn't always the correct choice for a programming project. Despite the real benefits it can confer upon a software development project, it can also have certain costs; these costs and benefits should be weighed against each other.

For instance, some types of software lend themselves more readily to abstracting with OO techniques than others. Object-oriented software development can sometimes take longer due to the overhead associated with its level of abstraction. OO software sometimes runs slower than other approaches; this has certainly been true in Perl, and although not usually a deal breaker, it is sometimes an important consideration. (Current work on the upcoming Perl 6 is addressing this performance issue.)

In spite of these strictures, OO programming is often an excellent choice; it has become a key approach to writing software in the Perl language.

3.1.2 Terminology

Object

An object is a collection of data that logically belongs together.

For instance, you might have a genome object that would have such attributes (or parts) as the name of the organism, the DNA sequence data, the start and end points for each exon, the genes with their associated lists of exons, and so forth. The exact nature of an object is a matter of logic and convenience, and in the end, it depends on the judgment

3.2 Using Perl Classes (Without Writing Them)

Before you actually start writing classes, it's helpful to know how to use them. This section shows you how to use OO Perl classes, even if the syntax is new to you and you've never written one yourself.

Thanks to the large and active community of Perl programmers, there are many useful Perl classes already written and freely available to use in your programs. Very often, the class you want already exists. All you need to do is obtain it and use it.

First, you need to find the appropriate module or modules (CPAN is the most common source for modules), install it, and examine the documentation to learn how to use the class. Finding and installing OO modules employs the same process covered in Chapter 1.

What's different about OO modules is how they create data structures and call and pass arguments to subroutines. In short, there's some new syntax to learn that amounts to a slightly different style of programming.

Object-oriented code creates an object by naming the class and calling a special method in the class (usually called new). The newly created object is a reference to a data structure, usually a hash. The object is then used to call methods (or OO subroutines). You're used to subroutines that get their data passed in as arguments; by contrast, OO code has a data structure that calls subroutines to operate on it. My goal in this section is to explain enough of this new terminology and syntax so you can read and understand the documentation for a class, and use it in your own programs.

Let's begin with the documentation for the Carp module, a nonOO module that appears later in this chapter. This is a simple module to use; it defines four subroutines, and the documentation gives brief examples of their use. Because the Carp module comes installed with any recent release of Perl, you don't have to install it. To find out how to use it, type:

```
perldoc Carp
```

(Upper- and lowercase is significant; typing perldoc carp won't work.) Here's the beginning of the output:

```
NAME
       carp    - warn of errors (from perspective of caller)
       cluck   - warn of errors with stack backtrace
                 (not exported by default)
       croak   - die of errors (from perspective of caller)
       confess - die of errors with stack backtrace

SYNOPSIS
       use Carp;
       croak "We're outta here!";

       use Carp qw(cluck);
       cluck "This is how we got here!";
```

It shows what subroutines are available and how to use them in your code. Additional details do appear in the documentation. They are important and sometimes necessary to read carefully, but you usually don't need to delve any further than the SYNOPSIS section that gives examples. To use the croak subroutine, you first load it with the directive use Carp;. You then call croak by providing a string containing a message as an argument; the program prints the message and dies.

3.3 Objects, Methods, and Classes in Perl

To the user of a class, the most important piece of information is the interface describing how to use the class. Usually this interface can easily be summarized in a few examples provided by the author of the class. The details of how the class is implemented may change; as long as the interface remains the same, your code needn't change even when the internals of the class you're using change. This provides good modularization, protecting your system from the ripple effects of changes in individual components, thus making your code as a whole more robust and reliable. This is one of the main benefits of OO design.

With OO design, you know that:

- A class is a package.
- An object is a reference to a data structure in the class, that is marked (or "blessed") with the class name.
- A method is a subroutine in the class.

Several other concepts and their associated terminology are also important in object-oriented programming. For instance, inheritance enables one class to use the definitions from another class, while adding to or changing the definitions.

At this point you may be wondering: if an object is really just a data structure (usually a reference to a hash), and if a method is really just a subroutine, then why all this new terminology? The answer is that the framework imposed upon these data structures and subroutines, in which each data structure has a defined set of subroutines that alone can access the data structure, is indeed a new level of abstraction—a new set of constraints influencing the programming structure.

These constraints have proved to be so frequently useful that the new terminology of class, object, and method does say something new about the data structures and subroutines involved. Also, there are some new features such as bless and the arrow notation that cause subroutines to behave a little differently, as has been mentioned already, and will be explained in detail. But surprisingly few such additions are needed to transition from a knowledge of Perl's declarative style to Perl's OO style.

When you program using a defined class with its methods and objects, you can gain access to the class data only with the class methods provided by the class designer. The restriction of access to a class's data to the methods alone is called encapsulation. From the standpoint of the programmer using the class, exactly how the methods and objects are implemented isn't necessarily a concern. As a programmer using the class, you can regard the class as a black box; you don't have to look inside to see how it is implemented. Usually, it's only the programer writing the code who needs to worry about that.

One final point: in the field of OO programming, different authors define terms in different ways and present differing sets of essential concepts. Be warned that considerable diversity exists in the literature and among the languages that deal with object-orientation. The excellent book Object Oriented Perl (see Section 3.14) includes a table that matches basic concepts with some of the alternate terminologies for those concepts.

3.4 Arrow Notation (->)

Object-oriented Perl code uses arrow notation (->) to call methods. Understanding how this works is essential to understanding OO Perl. Before you start reading OO Perl code, let's look more closely at its main features and how arrow notation is used to call methods.[2]

[2] The arrow (->) also appears in Perl when dealing with complex data structures, as you saw in Chapter 2, where it's used for references to subroutines.

The arrow notation is used on an object to call a method in a class. Because the object has been blessed (i.e., marked with the class name), Perl can tell from the object what class it's in and so knows to find the method in that same class. With arrow notation, Perl also passes to the method a new argument that automatically appears first in its argument list. The other arguments are shifted over; the first argument is now the second, and so on. The automatic passing of a new first argument to the method is the key to understanding OO Perl code.

The method name appears to the right of the arrow. Perl then uses what's immediately to the left of the arrow to identify the class in which to find the method. It also passes information about what's on the left of the arrow to the method, where it appears as the first argument of the method. The left side of the arrow may be in one of two forms:

- The name of the class. Here's an example:
  ```
  TRNA->new(  );
  ```
  Here Perl sees that the left side is the name of the TRNA class; therefore it calls the new subroutine in the TRNA package. It also automatically passes the name TRNA to that subroutine as its first argument, shifting any other arguments that may have been explicitly given (you'll see how this feature is used in later examples). You need to save the new object (a blessed reference to a hash) using the assignment operator = as follows:
  ```
  $trna_object = TRNA->new(  );
  ```
- An object. Here's an example:
  ```
  $trna_object->findloops(  );
  ```
  Perl sees that on the left side of the arrow $trna_object is an object of the TRNA class; it therefore calls the findloops method in the TRNA class. (It can see that $trna_object is a TRNA object because the object was blessed into the TRNA class when it was created by the new method, as I'll explain later.) Perl also passes a reference to the $trna_object object into the findloops method as the first argument to the method, shifting any other arguments that may have been explicitly given.

Why does Perl do it this way? The short answer to that question is that once you understand how it works, your code will become simpler and more usable. You will need to type class names less frequently, and you can use methods written for one class in another class (inheritance).

The two new tricks that Perl performs here are:

- Using arrow notation to find the correct method in the correct class
-

3.5 Gene1: An Example of a Perl Class

This first example of Perl code defines a very small Perl class. This code does several new things, which I will explain in detail after the code.

This first version of the class is called Gene1, and it demonstrates the essential features needed to implement a simple class. Gene1 looks similar to the last module definition in Chapter 1, but with a few new wrinkles that transform it into OO software. I progress from Gene1.pm to Gene2.pm, then to Gene3.pm, and then to the final version, Gene.pm.

The methods of the Gene1 class will permit creating Gene1 objects and finding out what the values of a Gene1 object's attributes are.

Here's the module that implements a Gene1 class. I put the module into a file called Gene1.pm and place it into a directory on my computer that can be found when Perl needs it. I continue putting my code into my own development library directory, which on my Linux system is the directory */home/tisdall/MasteringPerlBio/development/lib*. This directory is pointed out to Perl at the beginning of the testGene1 program that appears later as an example of how to use the Gene1.pm module definition. You will probably use a different directory on your computer, in which case you'll have to change this line:

```
use lib "/home/tisdall/MasteringPerlBio/development/lib";
```

You can also put the directory on the command line or set the PERL5LIB environmental variable, as described in Chapter 1. Setting the PERL5LIB variable is the easiest because you don't have to change the use lib lines in the programs.

A Gene1 object consists of a gene name, an organism represented by genus and species, a chromosome, and a reference to a protein structure in the PDB:

```
package Gene1;

use strict;
use warnings;
use Carp;

sub new {
        my ($class, %arg) = @_;
        return bless {
                _name       => $arg{name}        || croak("no name"),
                _organism   => $arg{organism}    || croak("no organism"),
                _chromosome => $arg{chromosome}  || "????",
                _pdbref     => $arg{pdbref}      || "????",
        }, $class;
}

sub name        { $_[0] -> {_name}     }
sub organism    { $_[0] -> {_organism} }
sub chromosome  { $_[0] -> {_chromosome}}
sub pdbref      { $_[0] -> {_pdbref}    }

1;
```

That's the whole thing!

3.6 Details of the Gene1 Class

In this section, I introduce the OO features used to make a class in Perl. First however, I explain the variable naming convention I use, as well as the handy Carp module.

3.6.1 Variable Names and Conventions

Using an underscore in front of a name is a programming convention that usually indicates that the item in question (e.g., a variable or hash key) isn't meant for the outside world but only for internal use.

This is just a convention; Perl doesn't require you to do it. It will, however, make your code easier to read and understand.

I generally follow this convention and put underscores in front of names that I don't want directly accessed by the programmer using the class. (In Perl, unlike some more strict OO languages, you can access data that's internal to a class, which make this naming convention that distinguishes internal variables particularly useful.)

Thus, in my Gene1 class, the attributes _name, _organism, _chromosome, and _pdbref are used internally only as the hash keys for the attributes in the object. When you use the class, as I do in my example program testGene1, you don't even have to know these names exist.

The interface is through arguments that specify the initialization values of these attributes. These arguments are called name, organism, chromosome, and pdbref. I also have methods—the subroutines also called name, organism, chromosome, and pdbref—that return the value of the actual attributes stored in the object.

3.6.2 Carp and croak

The Carp module is called near the top of Gene1.pm with use Carp;.

Carp is a standard Perl module that provides informative error messages in the case of problems. carp prints a warning message; croak prints an error message and dies. They are very much like the Perl functions warn and die; they report the line number in which the problem occured in the error message and report from what subroutine they were called. I use croak in my code; it prints out the error message provided, names the file and the line number and subroutine where it's called from, and then kills the program.

This function is certainly useful during development because it's another way to find errors in a program as it's being created. It also gives program users the ability to report the exact location of a problem, should one occur, to the programming staff (which may be just one programmer, you!).

In my program output, the Carp message is:

```
no name at testGene1 line 35
```

It's produced by the line:

```
_name      => $arg{name}     || croak("no name"),
```

3.7 Gene2.pm: A Second Example of a Perl Class

Gene1 demonstrated the fundamentals of a Perl class. Now, I'll build a more realistic example, which also includes a few additional standard Perl techniques.

My goal is to present an example that you can imitate in order to begin to develop your own OO software. I'm going to build the example in three more stages, expanding upon the Gene1.pm module. First, I'll add mutators, which are methods that alter the data in an object. I'll also add a method that gives information about the class as a whole, returning the count of how many objects in the class exist in the running program. This depends on the use of closures, methods that use variables declared outside the methods. This is the new material in the Gene2.pm module.

After that step, I introduce the AUTOLOAD mechanism, which gives a single class method called AUTOLOAD that can define large numbers of other methods and significantly reduce the amount of coding you need to write to develop a more complex object (among other benefits to be described later). That will be the Gene3.pm module.

We'll end up with a Gene.pm module you can use as a basis for your own Perl module development. It will add a mechanism to specify what properties each attribute has (which can prevent improper data manipulation, for instance). It will show how to initialize an object with class defaults and how to clone an existing object. Finally, Gene.pm will show you how to incorporate the documentation for a class right in the Perl code for the class.

Here is the code for the intermediate Gene2.pm module. Following the Gene2.pm module is an example of the code and output of a small test program that drives the module. Take a minute to look at these two code examples, especially at the comments. The module Gene2.pm contains several new details that will be discussed following the code. The test program should be fairly easy to read and understand.

```
package Gene2;

#
# A second version of the Gene.pm module
#

use strict;
use warnings;
use Carp;

# Class data and methods, that refer to the collection of all objects
# in the class, not just one specific object
{
    my $_count = 0;
    sub get_count {
        $_count;
    }
    sub _incr_count {
        ++$_count;
    }
    sub _decr_count {
        --$_count;
    }
}

# The constructor for the class
sub new {
    my ($class, %arg) = @_;
    my $self = bless {
        _name        => $arg{name}      || croak("Error: no name"),
        _organism    => $arg{organism}  || croak("Error: no organism"),
```

3.8 Gene3.pm: A Third Example of a Perl Class

We've gone through two iterations building an OO class with the Gene1.pm and Gene2.pm modules. Now, let's add a few more features and create the Gene3.pm module as a penultimate example for this introduction to OO programming in Perl.

Here is the code for Gene3.pm and for the test program testGene3; also included is the output produced by running testGene3. Following the code will be a discussion of the new features of this third version of our example class. But I'll point out before you read on, that AUTOLOAD is a special name in Perl for a subroutine that will handle a call to any undefined subroutine in a class. (I'll give more details after you look at the code.)

```
package Gene3;

#
# A third version of the Gene.pm module
#

use strict;
use warnings;
our $AUTOLOAD; # before Perl 5.6.0 say "use vars '$AUTOLOAD';"
use Carp;

# Class data and methods, that refer to the collection of all objects
# in the class, not just one specific object
{
    my $_count = 0;
    sub get_count {
        $_count;
    }
    sub _incr_count {
        ++$_count;
    }
    sub _decr_count {
        --$_count;
    }
}

# The constructor for the class
sub new {
    my ($class, %arg) = @_;
    my $self = bless {
        _name        => $arg{name}      || croak("Error: no name"),
        _organism    => $arg{organism}  || croak("Error: no organism"),
        _chromosome  => $arg{chromosome}|| "????",
        _pdbref      => $arg{pdbref}    || "????",
        _author      => $arg{author}    || "????",
        _date        => $arg{date}      || "????",
    }, $class;
    $class->_incr_count(  );
    return $self;
}

# This takes the place of such accessor definitions as:
#  sub get_attribute { ... }
# and of such mutator definitions as:
#  sub set_attribute { ... }
sub AUTOLOAD {
    my ($self, $newvalue) = @_;

    my ($operation, $attribute) = ($AUTOLOAD =~ /(get|set)(_\w+)$/);

    # Is this a legal method name?
```

3.9 How AUTOLOAD Works

The AUTOLOAD mechanism, built into the definition of Perl packages, is simple to use. If a subroutine named AUTOLOAD is declared within a package, it is called whenever an undefined subroutine is called within the package. AUTOLOAD is a special name, and must be capitalized as shown, because Perl is designed that way. Don't use the subroutine name AUTOLOAD (or DESTROY) for any other purpose, or you'll suffer unintended consequences.

Without an AUTOLOAD subroutine defined in a package, an attempt to call some undefined subroutine simply produces an error when the program runs. But if an AUTOLOAD subroutine is defined, it is called instead and is passed the arguments of the undefined subroutine. At the same time, the $AUTOLOAD variable is set to the name of the undefined subroutine.

Here's an example of a short Perl program that tries to call an undefined function:

```
#!/usr/bin/perl

use strict;
use warnings;

print "I started the program\n";

report_protein_function("one", "two");

print "I got to the end of the program\n";
```

It gives the following output:

```
I started the program
Undefined subroutine &main::report_protein_function called at jk.pl line 8.
```

Here's what happens when an AUTOLOAD subroutine is defined in the package:

```
#!/usr/bin/perl

use strict;
use warnings;
use vars '$AUTOLOAD';

print "I started the program\n";

report_protein_function("one", "two");

print "I got to the end of the program\n";

sub AUTOLOAD {
        print "AUTOLOAD is set to $AUTOLOAD\n";
        print "with arguments ", "@_\n";
}
```

It gives the following output:

```
I started the program
AUTOLOAD is set to main::report_protein_function
with arguments one two
I got to the end of the program
```

3.9.1 Defining Global Variables

Recall that when you start programs with such statements as:

3.10 Cleaning Up Unused Objects with DESTROY

When a running program no longer needs a portion of computer memory, what happens to it? How is the program's memory managed? Various possibilities exist, and different languages handle the problem in different ways. For instance, the designer of the language can just leave the memory as it is, unused, and go on and use other memory for other tasks. No clean up is strictly necessary.

However, this might, and does, cause problems with certain kinds of programs. Some programs read in large amounts of data into their memory, perhaps extract some statistics on the data, and then go on to the next large chunk of data to repeat the same operation. A computer's memory is finite; for a program that runs a long time and examines a continuous source of data (say, for instance, the data generated by your sequencing facility), it will at some point use all available main memory.

It is necessary to consider how to clean up memory that is no longer used, so it can be reused by the program. This is sometimes called the *garbage collection* problem. Consideration of this problem has resulted in many approaches and a large amount of literature, which won't be discussed here.

However, sometimes there are practical considerations. In the class module Gene.pm, I'm keeping count of all objects that are created by the running program. In Perl, when a variable is no longer used, its memory is automatically cleaned up. One such instance is when a variable goes out of scope. For instance, in the following code fragment, the variable $i goes out of scope after the if block, and its memory is cleaned up, making it available to the rest of the program:

```
if(1) {
        my $i = 'ACCGGCCGGCCGGTTAATGCATAATC';
        determine_function($i);
}

# $i has gone out of scope here
```

This problem actually affects the Gene.pm module. Say you create a new object, and as the program continues, the object goes out of scope. For instance, if the object was created within a block, and the program leaves the block, the object is then out of scope. Perl will remove the part of memory that held the object, and all will be well... except that the global count of the number of objects will now be off by one!

What is needed is a way to automatically call a bit of code to adjust the global count whenever an object goes out of scope. Perl provides such a mechanism with the DESTROY subroutine. Perl calls the DESTROY method 1) if you've defined a method with that name in your class, and 2) a class object (a reference blessed with the name of the class) goes out of scope. It does so automatically, just as AUTOLOAD is automatically called if you attempt to call a method that doesn't exist on a class object.

In our program, the only thing keeping track of when an object goes out of scope and is garbage collected by Perl is the global count of existing objects. This simple DESTROY subroutine will thus suffice:

```
sub DESTROY {
    my($self) = @_;
    $self->_decr_count(  );
}
```

Let's see if it works. Here's a test program, testGeneGC (GC for garbage collection):

```
#!/usr/bin/perl

#
```

3.11 Gene.pm: A Fourth Example of a Perl Class

We've now come to the fourth and final version of the Gene class, Gene.pm. This final version adds a few more bells and whistles to make the code more reliable and useful. You'll see how to define the class attributes in such a way as to specify the operations that are permitted on them, thus enforcing more discipline in how the class can be used. You'll also see how to initialize an object with class defaults or clone an already existing object. You'll see the standard and simple way in which the documentation for a class can be incorporated into the .pm file. This will conclude my introduction to OO Perl programming (but check out the exercises at the end of the chapter and see later chapters of this book for more ideas).

3.11.1 Building Gene.pm

Here then is the code for Gene.pm. Again, I recommend that you take the time to read this code and compare it to the previous version, Gene3.pm, before continuing with the discussion that follows:

```
package Gene;

#
# A fourth and final version of the Gene.pm class
#

use strict;
use warnings;
our $AUTOLOAD; # before Perl 5.6.0 say "use vars '$AUTOLOAD';"
use Carp;

# Class data and methods
{
    # A list of all attributes with default values and read/write/required properties
    my %_attribute_properties = (
        _name        => [ '????',        'read.required'],
        _organism    => [ '????',        'read.required'],
        _chromosome  => [ '????',        'read.write'],
        _pdbref      => [ '????',        'read.write'],
        _author      => [ '????',        'read.write'],
        _date        => [ '????',        'read.write'],
    );

    # Global variable to keep count of existing objects
    my $_count = 0;

    # Return a list of all attributes
    sub _all_attributes {
            keys %_attribute_properties;
    }

    # Check if a given property is set for a given attribute
    sub _permissions {
        my($self, $attribute, $permissions) = @_;
        $_attribute_properties{$attribute}[1] =~ /$permissions/;
    }

    # Return the default value for a given attribute
    sub _attribute_default {
            my($self, $attribute) = @_;
        $_attribute_properties{$attribute}[0];
    }

    # Manage the count of existing objects
    sub get_count {
        $_count;
```

3.12 How to Document a Perl Class with POD

An essential part of programming is documentation. Comments in the code are an important part of documenting code for those who have to read it or modify it in the future.

Equally as important is documentation for those who have to use the code. A short, accurate, practical guide to using a Perl class is absolutely necessary in order for the class to be generally useful.

Perl uses a language called POD (plain old documentation) to put documentation right in the code. The fourth and final version of Gene.pm has POD documentation embedded in it.

To gain access to the documentation, you merely have to type:

```
perldoc Gene.pm
```

in the same directory in which the Gene.pm lives. (For other options, see the perlpod manpage on the Web, or type perldoc perlpod.)

Given that this book contains copious amounts of explanation of the code, I've kept the POD documentation to a minimum. The POD language is simple; the best way to use it to write good documentation is to copy and modify the documentation style that's used by some other well-written module. You will almost always want to give a bit more information than the example shown here; try examining the documentation for some Perl modules on your computer, for example, perldoc CGI or, if it's installed, perldoc Bioperl.

The Perl interpreter will ignore everything from a line beginning:

```
=head1
```

up to a line beginning:

```
=cut
```

so you can embed your POD documentation in with your Perl code without difficulty.

It's also worth pointing out that many filters exist that will take your .pm file with its embedded POD documentation and produce versions of the documentation in HTML, LaTEX, plain text, *nroff*, or other formats.

Here's the output you get from typing perldoc Gene.pm:

```
Gene(3)        User Contributed Perl Documentation        Gene(3)

Gene
       Gene: objects for Genes with a minimum set of attributes

Synopsis
           use Gene;

           my $gene1 = Gene->new(
               name       => 'biggene',
               organism   => 'Mus musculus',
               chromosome => '2p',
               pdbref     => 'pdb5775.ent',
               author     => 'L.G.Jeho',
               date       => 'August 23, 1989'
```

3.13 Additional Topics

Included in this section are a few more topics you may find useful.

3.13.1 Using Class::Struct to Define Classes

The kind of simple OO class that I've developed in this chapter has proved so useful that some clever folks have written a Perl module Class::Struct that automates the construction of classes of this type.

It's worth examining Class::Struct because it can be a great timesaver for some situations. It's been used to create classes for many widely used modules. Type:

```
perldoc Class::Struct
```

to get the whole story.

3.13.2 Class Inheritance

An important part of OO programming deals with the use of one class to help define another. For instance, you may have a class Protein that defines attributes common to all proteins. You can then use the Protein class to define a new class ZincFingers, which perhaps would have all the attributes of the Protein class plus some additional attributes relevant to the study of zinc fingers.

You'll see the use of class inheritance in the next chapter.

3.13.3 Bioperl

Bioperl is a collection of modules of intense interest to the Perl bioinformatics programmer, written mostly in OO style. I'll take a look at the Bioperl software in Chapter 9.

3.14 Resources

A vast number of techniques are used in OO software development in Perl; many more than I have space to explore in this book. For more details than can fit into this book, I recommend these sources for more details on OO programming in Perl.

- Object Oriented Perl by Damian Conway (Manning Publishers). This is an excellent book and is useful for beginners to advanced. It even includes a few bioinformatics examples! My introduction to OO Perl has drawn gratefully on Conway's book. I urge readers who will be doing further OO Perl programming to get a copy. Some material is slightly dated; for example, the material on pseudohashes should be skipped.
- The perlobj page from the Perl documentation.
- The perlboot tutorial page from the Perl documentation is a beginning introduction to Perl objects.
- The perltoot tutorial page from the Perl documentation is a more detailed introduction to Perl objects.
- The perltootc tutorial page from the Perl documentation also includes more information on class methods.
- The perlbot tutorial page from the Perl documentation is a bag of tricks for Perl OO programming.
- Some books already mentioned in earlier chapters have extensive information about Perl OO programming, such as Programming Perl, Perl Cookbook, and Advanced Perl Programming.

3.15 Exercises

Exercise 3.1

Write brief descriptions of the main features of declarative programming, OO programming, logic programming, and functional programming. (See Section 3.14.)

Exercise 3.2

Give an example of a programming job that would be better with OO programming than with declarative programming.

Exercise 3.3

Give an example of a programming job that would be better with declarative programming than with OO programming.

Exercise 3.4

What bioinformatics problem might be best addressed with logic programming?

Exercise 3.5

Download and use a Perl class from CPAN.

Exercise 3.6

Write a Perl class that manages laboratory supplies.

Exercise 3.7

When would you want a separate initialization method for a class; when would you want the initialization to be part of the new constructor?

Exercise 3.8

Modify Gene.pm to keep count of how many objects refer to given organisms, chromosomes, authors, pdb references, and names.

Exercise 3.9

Add a DESTROY method to a class so an object can self-destruct.

Exercise 3.10

Beginning in the code for Gene3.pm you'll find the following regular expression:

```
    if($AUTOLOAD =~ /.*(_\w+)/) {
        $attribute = $1;
```

This only catches the last part of a name that has an underscore. What if you want to allow names such as get_other_var? Write a regular expression that would extract such names as other_var from get_other_var.

Exercise 3.11

In the code for Gene2.pm you'll find the following regular mutator method:

```
sub set_name {
    my ($self, $name) = @_;
    $self->{_name} = $name if $name;
}
```

This breaks if $name has certain values such as "", 0, or 0E0. How can you catch these cases?

Chapter 4. Sequence Formats and Inheritance

This chapter applies concepts and techniques from previous chapters to a concrete project: handling sequence files. The chapter also introduces a few new techniques including a very important one called class inheritance. The code developed in this chapter will also be incorporated into later chapters.

Class inheritance allows you to define a new class by inheriting from other classes—altering or making additions as needed. It's a style of software reuse that is particular to object-oriented design.

The first class developed in this chapter is a simple one: reading and writing files. Using inheritance, you can extend that class to a new one that can recognize, read, and write data in several different biological sequence datafile formats.

The goal is, as always, to learn enough about Perl to develop software for your own needs. The code in this chapter is designed with this goal in mind. In particular, the exercises at the end of the chapter will ask you to extend and improve the code in various ways.

4.1 Inheritance

You've seen the use of modules and how a Perl program can use all the code in a module by simply loading it with the use command. This is a simple and powerful method of software reuse, by which software can be written once but used many times by different programs.

You've also seen how object-oriented Perl defines classes in terms of modules, and how the use of classes, methods, and objects provides a more structured way to reuse Perl software.

There's another way to reuse Perl classes. It's possible for a class to inherit all the code and definitions of another base class. (This base class is sometimes called a superclass or a parent class.) The new derived class (a.k.a. subclass) can add more definitions or redefine certain definitions. Perl then automatically uses the definitions of everything in the old class (but treats them as if they were defined in this new derived class), unless it finds them first in the derived class.

In this chapter, I'll first develop a class FileIO.pm, and then use the technique of inheritance to develop another class SeqFileIO.pm that inherits from FileIO.pm. This way of reusing software by inheritance is extremely convenient when writing object-oriented software. For instance, I make SeqFileIO do a lot of its work simply by inheriting the base class FileIO and then adding methods that handle sequence file formats. I could use the same base class FileIO to write a new class that specializes in handling HTML files, microarray datafiles, SNP database files, and so on. (See the exercises at the end of the chapter.)

When inheriting a class, it is sometimes necessary to do a bit more than just add new methods. In the SeqFileIO class, I add some attributes to the object, and as a result the hash %_attribute_properties also has to be changed. So in the new class I define a new hash with that name, and as a result the old definition from the base class is forgotten and the new, redefined hash is used. As you read the new class, compare it with the base class FileIO. Make note of what is new in the class (e.g., the various put methods), what is being redefined from the base class (e.g., the hash just mentioned), and what is being inherited from the base class (e.g., the new constructor.) This can help prepare you to write your own class that uses inheritance.

Occasionally, you want to invoke a method from a base class that has been overridden. You can use the special SUPER class for that purpose. I don't use that in the code for this chapter, but you should be aware that it is possible to do.

4.2 FileIO.pm: A Class to Read and Write Files

Even though you can easily obtain excellent modules for reading and writing files, this chapter shows you how to build a simple one from scratch. One reason for doing this is to better understand the issues every bioinformatics programmer needs to face, such as how to organize files and keep track of their contents. Another reason is so you can see how to extend the class to deal with the multiple file format problem that is peculiar to bioinformatics.

It's not uncommon for a biologist to use several different types of formats of files containing DNA or protein sequence data and translate from one format to another. Doing these translations by hand is very tedious. It's also tedious to save alternate forms of the same sequence data in differently formatted files. You'll see how to alleviate some of this pain by automating some of these tasks in a new class called SeqFileIO.pm.

Class inheritance is one of the main reasons why object-oriented software is so reusable. In order to see clearly how it works, let's start with the simple class FileIO.pm and later use it to define a more complex class, SeqFileIO.pm.

FileIO is a simple class that reads and writes files, and stores simple information such as the file contents, date, and write permissions.

You know that it's often possible to modify existing code to create your own program. When I wrote FileIO.pm, I simply made a copy of the Gene.pm module from Chapter 3 and modified it.

On my Linux system, I started by copying FileIO.pm from Gene.pm and giving it a new name:

```
cp Gene.pm FileIO.pm
```

I then edited the new file FileIO.pm changing the line near the top that says:

```
package Gene;
```

to:

```
package FileIO;
```

The filename must be the same as the class name, with an additional .pm.

Though I now needed to modify the module to do what I want, a surprising amount of the overall framework of the code—its constructor, accessor and mutator methods, and its basic data structures—remains the same. Gene.pm already contained such useful parts as a new constructor, a hash-based object data structure, accessor methods to retrieve values of the attributes of the object, and mutator methods to alter attribute values. These are likely to be needed by most classes that you'll write in your own software projects.

4.2.1 Analysis of FileIO

Following is the code for FileIO, with commentary interspersed:

```
package FileIO;

#
# A simple IO class for sequence data files
#

use strict;
```

4.3 SeqFileIO.pm: Sequence File Formats

Our primary interest is bioinformatics.Can we extend the FileIO class to handle biological sequence datafiles? For example, can a class be written that takes a GenBank file and writes the sequence out in FASTA format?

Using the technique of inheritance, in this section I present a module for a new class SeqFileIO that performs several basic functions on sequence files of various formats. When you call this module's read method, in addition to reading the file's contents and setting the name, date, and write mode of the file, it automatically determines the format of the sequence file, extracts the sequence, and when available, extracts the annotation, ID, and accession number. In addition, a set of put methods makes it easy to present the sequence and annotation in other formats.[1]

[1] Don Gilbert's readseq package (see http://iobio.bio.indiana.edu/soft/molbio/readseq and ftp://ftp.bio.indiana.edu/molbio/readseq/classic/src) is the classic program (written in C) for reading and writing multiple sequence file formats.

4.3.1 Analysis of SeqFileIO.pm

The first part of the module SeqFileIO.pm contains the block with definitions of the new, or revised, class data and methods.

The first thing you should notice is the use command:

```
use base ( "FileIO" );
```

This Perl command tells the current package SeqFileIO it's inheriting from the base class FileIO. Here's another statement that's often used for this purpose:

```
@ISA = ( "FileIO" );
```

The @ISA predefined variable tells a package that it "is a" version of some base class; it then can inherit methods from that base class. The use base directive sets the @ISA array to the base class(es), plus a little else besides. (Check perldoc base for the whole story.) Without getting bogged down in details use base works a little more robustly than just setting the @ISA array, so that's what I'll use here:

```
package SeqFileIO;

use base ( "FileIO" );

use strict;
use warnings;
#use vars '$AUTOLOAD';
use Carp;

# Class data and methods
{
    # A list of all attributes with defaults and read/write/required/noinit properties
    my %_attribute_properties = (
        _filename    => [ '',   'read.write.required'],
        _filedata    => [ [ ],  'read.write.noinit'],
        _date        => [ '',   'read.write.noinit'],
        _writemode   => [ '>',  'read.write.noinit'],
        _format      => [ '',   'read.write'],
        _sequence    => [ '',   'read.write'],
        _header      => [ '',   'read.write'],
        _id          => [ '',   'read.write'],
        _accession   => [ '',   'read.write'],
    );
```

4.4 Resources

- Inheritance is a fundamental OO technique; see Section 3.14 for Perl OO reference material that includes discussion of inheritance.
- For programming with sequence data file formats, see the C program *readseq* by Don Gilbert (at http://iobio.bio.indiana.edu/soft/molbio/readseq).
- See the Bioperl project at http://www.bioperl.org for alternate ways to handle this programming task in Perl.
- For a more rigorous but slower approach to parsing sequence files (which is sometimes what you want) see the module Parse::RecDescent by Damien Conway at CPAN.
- Each sequence file format has documentation that describes it. These formats sometimes change to keep up with the changing nature of biological data. One of the following exercises challenges you to find the documentation for one of the formats and to improve the code in this chapter for that format.

4.5 Exercises

Exercise 4.1

Write an object-oriented module DNAsequence whose object has one attribute, a sequence of DNA, and two methods, get_dna and set_dna. Start with the code for Gene.pm, but see how far you can whittle it down to the minimum amount of code necessary to implement this new class.

Exercise 4.2

The FileIO.pm module implements objects that read and write file data. However, they can, depending on the program, deviate substantially from what are actually present in files on your computer. For instance, you can read in all the files in a folder, and then change the filenames and data of all the objects, without writing them out. Is this a good thing or a bad thing?

Exercise 4.3

In the text, you are asked why the new constructor for FileIO.pm has been whittled down to the bare bones. You can see that all it does is create an empty object. What functionality has been moved out of the new constructor and into the read and write methods? Does it make more sense to do without a new constructor entirely and instead have the read and write methods create objects? Try rewriting the code that way. Alternately, does it make sense to try rewriting the code so that both reading and writing are handled by the new constructor? Is creating an object sometimes logically distinct from initializing it?

Exercise 4.4

Use FileIO.pm as a base class for a new class that manages the annotation of a pipeline in your laboratory. For example, perhaps your lab gets sequence from your ABI machine, screens it for vectors, assesses the quality of the sequencing run, searches your local database to determine if you've seen it or something like it before, then searches GenBank to see what other known sequences it matches or resembles, and finally adds it to an assembly project. Each step has a person or persons, a timestamp for the beginning and ending of each phase, and data. You want to be able to track the work done on each sequence that emerges from your ABI. (This is just an example. Pick a set of jobs that you actually do in your lab.)

Exercise 4.5

For each sequence file format handled by the SeqFileIO.pm module, find the documentation that specifies the format. Compare the documentation with the is_, parse_, and put_ method to recognize, read, and write files in each format. How can you improve this code? Make it more complete? Faster?

Exercise 4.6

My parse_ methods are somewhat ad hoc. They don't really parse the whole file according to the definition of the format. They just extract the sequence and a small amount of annotation. Take one of the formats and write a more complete parser for it. What are the advantages and disadvantages of a simple versus a more complete parser in this code? How about for other applications you may want to develop in the future?

Exercise 4.7

Use the parser you developed in Exercise 4.6 to do a more complete job of identifying a file in the same format in the module's is_ method.

Exercise 4.8

Add a new sequence file format to SeqFileIO.

Exercise 4.9

In FileIO.pm, and in many other places in this book, the program calls croak and exits when a problem arises (such as when unsuccessfully attempting to open a file for reading). Such drastic measures are sometimes desirable; for example, you may want to kill the program if a security problem is discovered in which someone is attempting to read a forbidden file. Or, when developing software, you may like your program to print an informative message and die when a problem occurs, as that might help you develop the program faster.

Chapter 5. A Class for Restriction Enzymes

In this chapter, you'll learn how to write an object-oriented class that handles restriction enzymes using the modules from the previous chapter as part of an interface to the Restriction Enzyme Database (Rebase). I'll develop a class that finds restriction sites in DNA sequence data. In my book Beginning Perl for Bioinformatics, I presented code that extracts information from Rebase and uses it to make restriction maps of DNA sequence data. In this chapter, I'll adapt and extend that software (or ideas from the accompanying exercises) in an object-oriented fashion. (All code is shown here and is available from this book's web site.)

5.1 Envisioning an Object

The Rebase project provides a set of files that specify restriction enzymes, their cut sites, and a great deal more information. Consider the problem of designing an object-oriented version of code that uses this data. What will be the objects and the methods?

Each restriction enzyme has a name; associated with its name are the definition of its recognition site (which I'll translate into a Perl regular expression), information about the chemistry of the restriction enzyme, vendors of the enzyme, and other annotation. This information is all part of the Rebase database.

Perhaps I should consider each restriction enzyme as a suitable candidate for my basic object. I can then read in the Rebase database, creating objects for each restriction enzyme that includes such attributes as the recognition site, the translation of the recognition site into a Perl regular expression, and whatever additional annotation I find useful.

With such objects, I can associate methods that take as their arguments sequence data and return the list of locations in which that particular enzyme has a recognition site in the sequence. Sounds good, let's start coding!

But wait. What happens if, as is often the case, you want to find multiple restriction enzymes in a sequence and display the resulting map. With my design, you'd have to find the object associated with each restriction enzyme, pass it to the sequence, collect the locations, and then combine the individual lists of locations in order to display the map. This can be slow (finding the right objects, one for each restriction enzyme) and inconvenient (combining the output of the various methods from the various objects).

You recognize this questioning as an essential step in program design—thinking about the problem and considering alternative ways to write code that solve it. I reprise the idea here because, so far, I've been simply seeing and discussing solutions. Although it's neat and tidy, it isn't really the way programming works. Programming often involves thinking of alternative program strategies, comparing them, coding the most promising alternatives as prototypes and testing them (i.e., benchmarking), and finally deciding on an approach to implement.

So, in that spirit, what alternatives come to mind to the one enzyme/one object approach just described? The Rebase database is essentially a key/value lookup database, in which the key is the enzyme name. The value is the recognition site or annotation: actually there are several datafiles provided in the database. But I'm most interested in getting the recognition site, translating it to a Perl regular expression, and reporting on the locations in some sequence data. A nice interface to display some of the annotation of the restriction enzyme would also be useful.

Any key/value type of data immediately brings the hash data structure to the mind of the Perl programmer. As you know from my introduction to object-oriented programming, the hash data structure is also the most useful way to implement an object.

So, perhaps instead of many objects, one for each restriction enzyme, you may want to consider one object that provides the fast lookup of a value (the recognition site and regular expression) for each key (the name of the restriction enzyme). Clearly, this can be implemented as a hash. Other attributes can hold the sequence and the map as an array of the positions in the sequence in which the recognition sites exist. Methods for the object could extract the site, the regular expression, and perhaps some annotation, for each enzyme. A method can also locate the recognition sites for an enzyme in the sequence.

5.2 Rebase.pm: A Class Module

Here is a very simple interface to the Rebase data contained in the bionet file that is part of its distribution:

```
package Rebase;

#
# A simple class to provide access to restriction enzyme data from Rebase
#  including regular expression translations of recognition sites
#

use strict;
use warnings;
use Carp;
use DB_File;

# Class data and methods
{
    # A hash of all attributes with default values
    my %_attributes = (
        _rebase      => { },
             #    key   = restriction enzyme name
             #    value = space-separated string of sites => regular expressions
        _bionetfile  => '??',
        _dbmfile     => '??',
        _mode        => 0444,
    );

    # Return a list of all attributes
    sub _all_attributes {
        keys %_attributes;
    }

    # Return the value of an attribute
    sub _attribute_value {
        my($self,$attribute) = @_;
        $_attributes{$attribute};
    }
}
```

Notice that the opening block is considerably pared down, compared to earlier classes. For instance, I've tossed the code that keeps count of all objects. Why? Because it's unlikely that more than one of these objects will be necessary in a program: so why bother?

5.2.1 Attributes: Short and Sweet

Notice that the list of attributes is short:

_rebase

A hash that will be populated to provide the lookup, with enzyme names for keys, and recognition sites (and their translation to regular expressions) for values. (Make sure you see how in the hash %_attributes the value of the key _rebase is itself an anonymous hash.)

_bionetfile

The name of the datafile from the Rebase distribution. In my examples, I use the version numbered bionet.212, and by the time you read this book, more recent versions will be available (you can get bionet.212 from this book's web site).

_dbmfile

5.3 Restriction.pm: Finding Recognition Sites

The time has now come to write the class that creates an object out of a sequence, enzyme name(s), and the map of the location(s) of the enzyme recognition sites in the sequence.

This module depends a great deal on the module Rebase developed in the previous section, but it is fairly short because it just tries to do a small job. This new Restriction class takes a Rebase object (which has the Rebase database translated to regular expressions), some sequence, and a list of enzymes, and uses the regular expressions to find the recognition sites in the sequence. (Note that it doesn't use inheritance; it simply creates a Rebase object to use.)

In this module, the restriction map (the list of locations where the enzymes have recognition sites in the sequence) is obtained through an auxiliary method map_enzyme that simply lists the locations. Clearly, a more graphical display would be easier and more useful. I'll consider that as the book progresses.

5.3.1 The Restriction.pm Module

Here is the Restriction.pm module:

```
package Restriction;

#
# A class to find locations of restriction enzyme recognition sites in
#  DNA sequence data.
#

use strict;
use warnings;
use Carp;

# Class data and methods
{
   # A list of all attributes with default values.
   # "enzyme" is given as an argument possibly multiple time, set as key to _map hash
    my %_attributes = (
        _rebase      => { },  # A Rebase.pm hash-based object
        # key   = restriction enzyme name
        # value = space-separated string of recognition sites => regular expressions
        _sequence    => '', # DNA sequence data in raw format (only bases)
        _map         => { },# a hash: keys are enzyme names,
                            # values are arrays of locations
        _enzyme      => '', # space- or comma-separated enzyme names,
                            # set as key to _map hash
    );

    # Global variable to keep count of existing objects
    my $_count = 0;

    # Return a list of all attributes
    sub _all_attributes {
        keys %_attributes;
    }

    # Manage the count of existing objects
    sub get_count {
        $_count;
    }
    sub _incr_count {
```

5.4 Drawing Restriction Maps

One of the most important lessons of scientific programming is the importance of a good display of a program's results.

In this section, I'm going to add the ability to output a restriction map by inheriting Restriction.pm and making a new derived class Restrictionmap.pm. The restriction map will be shown very simply as the sequence printed in lines of simple text. The locations of restriction sites are written over the lines of text, giving the names of the restriction enzymes at the restriction sites. In Chapter 8, this simple text-based graphic output is replaced by a real picture (with colors, different fonts, and whatever bells and whistles you choose to add). I designed my base class Restriction.pm to represent the restriction map as a simple list of recognition site locations because I wanted my software to be flexible enough to be extended to accommodate any of the many different graphic formats that might be desired.

The difference between an unreadable mass of data, and a good clean graphic display that leads the eye towards an interesting result, is profound. It's the difference between a successful program and a dud, between hours or days spent sorting through columns of data and a quick discovery of a region of interest. It's even the difference between a scientific discovery, and none at all.

Graphics programming is a bit of an advanced topic. (You'll get your feet wet in Chapter 8.) But even if you need a program that's restricted to text output, you still need to spend programming time displaying that output in a useful manner. So, in this section, I'll add a very simple map drawing capability to the software, drawn as simple text.

5.4.1 Storing Graphics Output in an Attribute

I want to display a graphic. Do I need to add _graphic and _graphictype attributes to my object? The question boils down to: shall I compute and display a graphic whenever needed, or shall I compute a graphic and store it in an attribute? If you're new to computer graphics, just think of a graphic as a mass of data, which can be stored in a variable, or in a file on disk, and can be used to generate a graphical display on the computer screen by the right software.

I do already compute the restriction map, by which I mean the actual locations of the recognition sites for each enzyme requested, and stored it in the _map attribute. I can also compute a graphic at the same time and store it in the proposed _graphic attribute.

Let's think about the pluses and minuses of storing graphics output in an attribute.

I'm not sure which graphics system I'll use in the future; for now, a simple text output may work as a stored attribute, but there are a lot of graphics outputs possible. Also, it seems likely that I'll be able to compute the graphics output very quickly, so the need for storing it is less compelling. And storing a large image for possibly hundreds or thousands of objects will be a strain on the computer system.

However, I may not be able to calculate quickly for fancier, full color, high resolution graphics that I might want in the future. And perhaps I'll need to flip between graphics very quickly, in which case having them precalculated would be a necessity!

5.5 Resources

- The primary resource for this section is the Restriction Enzyme Database web site found at http://www.neb.com/rebase.
- See the discussion of making restriction maps in O'Reilly's Beginning Perl for Bioinformatics.

5.6 Exercises

Exercise 5.1

Why use the object-oriented approach for the interface to the Rebase database at all? What are the benefits and detriments of going to the object-oriented style?

Exercise 5.2

The Restriction.pm module uses another module in a new way. Instead of inheriting the Rebase.pm class, it requires that a Rebase object be passed to the constructor Restriction->new to become one of the attributes of the Restriction object.

Consider alternative ways to write this code. Can Restriction inherit Rebase and achieve the same functionality? If so, write the code. Or, can the same functionality be achieved by some method that avoids having a Rebase object passed as an argument to a Restriction object? If so, write the code.

Exercise 5.3

Go to CPAN and read the documentation about the MLDBM module. It allows you to use a DBM file to store and retrieve complex data. Rewrite the Rebase.pm module to use MLDBM and replace my use of space-separated strings of recognition sites and regular expressions.

Exercise 5.4

As discussed in the text, there are some interesting considerations involved in parsing the data that relates to how the restriction enzymes actually work, such as handling reverse complements of recognition sites and cut sites. The logic used here to handle reverse complements might not be ideal for all situations. Review carefully the logic of the parse_rebase subroutine. Can you find any problems its logic might cause when you try to use the software to support a particular experiment?

Exercise 5.5

It would be nice to be able to ask some method in Restriction.pm if a particular restriction enzyme produces sticky ends at its cut site. It would also be useful to know what other enzymes create sticky ends that will anneal with the sticky ends of this enzyme. Check to see if this information appears in any of the datafiles of the Rebase database. Can you design a method that returns this information, given the name of a restriction enzyme? What changes do you have to make to your database; do you need any more datafiles from the Rebase distribution?

Exercise 5.6

Describe in detail how the logic for map_enzyme works. Can you devise a different way to accomplish the same thing?

Exercise 5.7

The code in this chapter uses the class Restriction as a base class for the class Restrictionmap which lets you make a graphic display of the restriction map. Would it be a better idea just to add the graphics capabilities to the Restriction class instead of inheriting it into a new class? Rewrite Restriction to add the graphics capability to it. What are the pros and cons of these two different ways of writing and organizing the code?

Exercise 5.8

In the method _formatrestrictionmap, some lines of code are commented out that shorten the output by not printing extra blank lines. Try it out both ways. (And may God have mercy on your souls.) Do you think it makes the output less lengthy at the expense of making it more difficult to read? What is the tradeoff here? Do you prefer the longer or shorter version? Defend your preference.

Exercise 5.9

Add position numbers to the output of Restrictionmap. Add the position of the first base in each line or the position of each restriction enzyme.

Exercise 5.10

Part II: Perl and Bioinformatics

Chapter 6. Perl and Relational Databases

Relational database systems are extremely important in all kinds of computing—commercial as well as scientific. The Perl programmer can perform most database manipulations from Perl programs using modules written for this purpose. I'll briefly review database lore and then concentrate on an introduction to the Perl modules that provide an interface to relational databases.

This and the remaining chapters of this book will continue to look at fundamental Perl topics but with this difference: these topics rely on Perl modules, not on new Perl syntax. The reason for this is a deliberate decision by the Perl language designers to keep the language itself fairly small and to move as much functionality as possible into modules. This decision is interesting and important, and it has the practical effect of making modules quite important in Perl.

First, I'll provide a quick explanation of database terminology and acronyms. Since I'll be discussing only relational databases, I sometimes say database to mean relational database. (They aren't synonymous in general, however.) I say DBMS (or database management system(s)) to refer to the software that provides database capabilities, such as MySQL or Oracle. Database or relational database refers to the definition and implementation of a particular collection of data in a DBMS, such as the examples I show later in the chapter. These terms, database for the data itself, and database management system for the software system that handles the data, are often used informally and interchangeably, and it's usually clear what is meant.

6.1 One Perl, Many Databases

There comes a time when disk files or the simple DBM hash database (that you've seen in previous chapters) just won't manage the data of a medium- or large-size project, and you must turn to relational databases. Although they take quite a bit more effort to set up and to program, they offer a standard and reliable way to store data and to ask questions about it.

There are two things that make relational databases standard. For one thing, they all follow a certain model of data structures, the relational model. These data structures have become a fixture in the computing world; they combine a level of constraint and flexibility that has proved its usefulness in many areas, including bioinformatics.

Almost all relational databases are programmed with a programming language called the Structured Query Language, or SQL. This is a fairly simple language that creates, populates, queries, and manages the kind of data structures relational databases provide. The combination of a standard data structure with a standard programming language is another reason relational databases have become so successful.

One thing that's not standard is the proliferation of relational database companies and their penchant for doing things their own way. This may sometimes be a marketing decision, but it's more often the natural process of evolution—of different sets of programmers having different ideas and making different implementations.[1]

[1] When I first released software that used Perl for bioinformatics, I received a letter arguing that because C was available everywhere and Perl wasn't, Perl for bioinformatics was therefore a Bad Idea and I should use C instead. Of course, Perl is available everywhere now, including on the VMS systems that my correspondent was complaining about, and bioinformatics software is written in a variety of languages. He made the classic mistake of wanting to standardize a field long before it had settled down.

This is important when you have some working database application that uses a particular DBMS such as Oracle, and you find that you have to port the application to work on another DBMS such as MySQL. Perhaps another database system has become significantly faster or cheaper, or your computer is replaced with a new one that supports a different database, or your computer center or CIO decrees that some new DBMS is now the mandatory standard. If your database application makes extensive use of a feature that is available only on your old DBMS, you'll have a lot of work ahead of you rewriting your software to make it work on the new DBMS.

Luckily, thanks to some expert Perl programming, there is a way to get around this proliferation of different DBMS with their special ways of doing things and their special extensions of SQL. In this chapter, I'll use the Perl DBI (DataBase Independent) module that provides a common interface to different relational database systems; it makes it possible to write SQL that will run on many different relational database systems with little or no change.

Still, unless you are subject to a decree, the problem that the Perl bioinformatics programmer faces at the beginning of a project is, "Which relational database system should I use?" It depends on the computer you're on and what DBMS is already in use, available, paid for, or known locally. There are very expensive systems, and there are free ones: we'll take a quick look at some of the alternatives and use one of the most popular free ones for the following examples.

The beginning programmer should be aware that relational databases are a large field of endeavor. Stop at any local bookstore with a good computer book section and you'll see an impressive number of books dedicated to relational databases and SQL in general, and especially dedicated to working with specific relational database management

6.2 Popular Relational Databases

Relational database management systems are big business. They account for a significant segment of the computer business. Large companies often use them to manage their internal affairs and their sales data. Universities manage their internal affairs, and their research projects with them. Various governments use them extensively, from the national to the local levels.

The industry leader at present for high-end systems is Oracle. Along with other DBMS vendors such as IBM, Microsoft, Sybase, Informix, and more, they provide large database systems that can handle large amounts of data, and many queries against that data, very quickly. They also provide design and management tools for the programming and support staff a large institution needs to maintain a database system.

Unfortunately, these very nice software systems are also very expensive. They have hefty price tags themselves, and they require high-end computer systems to run on.

From the top-of-the-line systems on down, there are many vendors and price ranges in the database marketplace.

The main contenders in the free DBMS marketplace are mSQL, MySQL, and PostgresSQL. These systems are all progressing steadily, even rapidly, in their abilities. I use MySQL in this book, but the reasons aren't terribly important. The code I write in Perl and SQL runs with little need for change on any of these systems. If you're in the position of actually having to decide on an DBMS to install on your computer, you can find the information you need on the home web pages for these systems.

MySQL is my DBMS of choice because it's free; suitable for small- to medium-size database projects; and runs on most operating systems found in the lab, such as Mac, Windows, and Unix/Linux. It lacks some features major systems have, but it has enough of them and is implemented well enough that many businesses and many research laboratories have found it quite suitable for their work. Approximately three million servers have it installed. On balance, it has very good performance.[2]

[2] MySQL's multithreading helps account for its good performance, but it lacks some of the more advanced features of other DBMS available. Competition between these DBMS is keen, however, and there has been a certain amount of jockeying for bragging rights as performance and feature sets are improved.

The details of MySQL are available, and you can get a free copy of it from http://www.mysql.org.

6.3 Relational Database Definitions

A relational database is essentially a set of tables (*relations*) that are interrelated in certain ways. A table is a two-dimensional matrix with a name. Each table is composed of rows (*tuples*), and no two rows can have exactly the same values. Each row is composed of named fields (*attributes*); each field has a certain data type and a name, and a field can only contain one value.[3]

[3] As the name "relation" suggests to the mathematically inclined, each table represents a subset of the Cartesian product of the domains of the fields, in which each row is an element of the relation. However, the order of the fields in a table is not significant (because each field has a unique name), and that's an important departure from the standard set-theoretic definition of a relation.

For instance, a table may be defined with the fields Name as a character string of at most 50 characters, an ID as an integer, and a Date as a special date datatype. Each row then has a Name, ID, and Date value. Table 6-1 and Table 6-2 show a table called genename with three fields and three rows, and one called organism with two fields and five rows.

Table 6-1. genename

Name	**ID**	**Date**
aging	118	1984-07-13
wrinkle	9223	1987-08-15
hairy	273	1990-09-30

Table 6-2. organism

Organism	**Gene**
human	118
human	9223
mouse	9223
mouse	273

6.4 Structured Query Language

The Structured Query Language (SQL, pronounced "s q l" or "see quel") can be thought of as the working definition of a relational database. It provides the bioinformatics programmer with the wherewithal to create, populate, interrelate, query, and update a relational database on a computer system. Your DBMS comes with its own implementation of SQL, which will have all the basic commands plus some, or maybe all, the less-used commands and features in the standard definition of the language, perhaps even some special extensions to the standard.

SQL dates back to the 1970s when it was developed at IBM. The most widely used versions of SQL are based on the standard published in 1992 and commonly called SQL2. A newer standard called SQL3 is available and supports emerging database functionality such as object-oriented and object-relational data models. MySQL is based on a subset of the most commonly used parts of SQL2, with the goal of providing a very fast implementation of the key components of SQL. Some features of SQL3 are also being added.

SQL is actually a fairly simple language to learn. Most people find that getting an account established on their computer, reading through a quick tutorial, and then having example code to copy and modify with the SQL documentation close at hand, is enough to get started writing useful SQL code.

I'm not going to present an extensive SQL tutorial here, for three reasons. First, such tutorials are easily and widely available. Second, each DBMS has its own version of SQL, so the DBMS documentation (such as that which comes with MySQL, for example) is necessary and available to you anyway. Third, SQL is such a basically simple language that it's quite useful to learn the basics of it by simply seeing a few examples. That's the approach I'll take.

If you are new to SQL, the best way to get familiar with it is by using the interactive command-line interface to try out different commands. The following section demonstrates my Linux system running MySQL.

6.4.1 SQL Commands

First, I enter the interactive mysql program, providing my MySQL username ("tisdall") and interactively entering my MySQL account password:

```
[tisdall@coltrane tisdall]$ mysql -u tisdall -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.41

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Next, I ask for a list of all the databases that are defined in my MySQL DBMS:

```
mysql> show databases;
+----------+
| Database |
+----------+
| caudyfly |
| dicty    |
| gadfly   |
| master   |
| mysql    |
| poetry   |
| yeast    |
+----------+
7 rows in set (0.15 sec)
```

6.5 Administering Your Database

Database administration encompasses such tasks as installing and configuring the DBMS, backing up the data, adding users and setting their various permissions, applying updates or new capabilities to the system, and similar tasks. If you just have yourself and a fairly small lab to deal with, it's not too bad. But organizations often hire one or more database administrators to do this work full time; even a smallish project, if it's critical and the budget exists, can benefit from the attention of a professional database administrator.

If you are a beginning programmer and need to install and maintain a database management system, you'll need to read the manuals and learn the tools. Even if you have some computer administration experience, there is a bit of learning involved. The best thing you can do is get help from an experienced database administrator.

Failing such expert help, it's necessary to get good documentation and follow it. This depends on the system you're using, of course. The following sections describe some of the basics.

6.5.1 Adding Users

One function a database administrator needs to know is how to add users and set their permissions, that is, what operations they're allowed to perform, and what resources they're allowed to view or change. In MySQL, for example, each user needs an account name and a password for access (these aren't tied to the rest of the account names and passwords on the computer system). Security can be important as well. You may use the database to manage your new data and results, which you don't want to release to the public just yet; at the same time, you may be providing the public, through a web site, access to your more established data and results. A system such as MySQL provides several tools to set up and manage accounts and security permissions.

6.5.2 Backup and Reloading

One essential task for any computer system's effort, including working with databases, is to back up your work. All computers will break; every disk drive will crash and become inoperable. If you don't have timely backups of your data, you will certainly lose it.

There are many ways to back up data; even MySQL has more than one method. However, even if you back up your data from the database to a backup file, it's still necessary to make a copy of the backup file in another location that's not on the same hard disk. For the small to medium project, it's possible to run a program that simply makes a text file containing MySQL commands that repopulates your database. This is often a convenient and workable method, but check the MySQL documentation if you wish for alternatives.

Here, then, is how you can make a backup, or dump, of a database, in this case to the disk file homologs.dump:

```
 [tisdall@coltrane development]$ mysqldump homologs -u tisdall -p > homologs.dump
Enter password:
[tisdall@coltrane development]$
```

After that command, a dump file called homologs.dump is created. Here's what it looks like for my little two-table database:

```
 [tisdall@coltrane development]$ cat homologs.dump
# MySQL dump 8.14
#
# Host: localhost    Database: homologs
#
```

6.6 Relational Database Design

Database design is the process of effectively organizing data into tables in a relational database. When thinking about how to organize a database you need to ask, "What fields should I put together into tables, and how should I interrelate the tables?" In this section I will show you a short example that demonstrates some common and useful techniques for doing just that.

Relational database software projects are best broken down into separate stages. This list from Database Systems: a Practical Approach to Design Implementation, and Management (see Section 6.10), shows the typical stages of database design and construction:
Database planningSystem definitionRequirements collection and analysisDatabase designDBMS selectionApplication designPrototypingImplementationData conversion and loadingTestingOperational maintenance

For the small biology lab in which database programming may be a one-person project, some of these stages may be brief and informal, but they still apply.

How should the tables be defined for a new database? The answer depends on the problems to be answered by the data, but it's also largely a matter of common sense and a feel for the data. The database beginner typically looks at the data, tries her hand at a few designs, and begins to get a sense of how tables can be used for a specific problem. Let yourself experiment and try a few alternatives, and you'll soon get the hang of it.

Tables are commonly interrelated by indexing and by joining fields from different tables. The SQL language implemented with your DBMS provides these abilities. Also, a group of techniques called *normalization* can help you produce a good design and avoid some problems. Simply putting data into tables doesn't guarantee a good design.

A set of rules called normal forms helps you arrange the data into tables in a way that avoids certain problems. One such problem is data redundancy, which is unnecessary duplication of data in different tables. A related problem is update anomalies caused by having the same data in more than one location. When such data is updated, copies may not be updated properly, and the database can become inaccurate.

Here are some simple rules to follow when designing your database:

- Each entry of each table has a single value. This is first normal form.
- Each table has a a unique identifier (called a primary key) for each row. This is second normal form.
- Names aren't used as identifiers because they can lead to data redundancy.

Third and other normal forms as well as other design considerations aren't covered in this book due to space limitations. See Section 6.10 at the end of the book for more information about relational database design. Consider Table 6-3, which shows this alternate, unnormalized version of my homologs database.

6.7 Perl DBI and DBD Interface Modules

SQL is a fairly simple and easy-to-learn language, considered by most to be well-tailored to its task. However, there are things available in most programming languages, such as control flow (while, for, foreach) and conditional branches (if-else) that aren't provided in most implementations of the language. The lack of these abilities severely restricts the use of SQL as a standalone language.

Most applications that use a relational database are written in another language such as Perl. Perl provides a link between the application, the programmer, the user, the files, the web server, and so on. Perl also provides the program logic. The interaction between the application and the database is typically to execute some database commands, such as fetching data from the database and processing it using Perl's capabilities. The logic of the program may depend on the data found in the database, but it is Perl, not SQL, that provides this logic (for the most part).

In Perl, a set of modules have been written that allow interaction with relational databases. The DataBase Independent (DBI) module handles most of the interaction from the program code; the DataBase Dependent (or DataBase Driver) (DBD) modules, different for each particular DBMS, handle communicating with the DBMS.

6.7.1 Installing and Configuring Perl DBI and DBD Modules

To use a MySQL database from Perl, you need first to have installed and properly configured MySQL. This is not a Perl job, but a database administration job; you have to get MySQL and install it on your system and set up the appropriate user accounts and permissions.

You have to then install the Perl DBI module (http://www.symbolstone.org/technology/perl/DBI) using CPAN from the command line:

```
perl -MCPAN -e shell;
```

Then type:

```
install DBI
```

You can also install it by downloading the module from CPAN, for example, via a web browser and following the QUICK START GUIDE instructions shown here:

```
QUICK START GUIDE:

    The DBI requires one or more 'driver' modules to talk to databases.

    Check that a DBD::* module exists for the database you wish to use.

    Read the DBI README then Build/test/install the DBI by doing
            perl Makefile.PL
            make
            make test
            make install
    Then delete the source directory tree since it's no longer needed.

    Use the 'perldoc DBI' command to read the DBI documentation.

    Fetch the DBD::* driver module you wish to use and unpack it.
    http://search.cpan.org/ (or www.activestate.com if on Windows)
    It is often important to read the driver README file carefully.
    Generally the build/test/install/delete sequence is the same
    as for the DBI module.
```

6.8 A Rebase Database Implementation

In earlier chapters, I developed an object-oriented interface to Rebase that stores the data in a simple DBM hash database, and in this chapter I showed how to interact with a MySQL relational database management system.

Now, let's put the two things together.

In this section, I'll take the Rebase.pm module that implements the Rebase class and modify it to use a relational database instead of Perl's simple hash-based DBM database. I'll call the resulting class RebaseDB.

For a small problem like this, either approach works pretty well. In fact, on my computer, the DBM approach is considerably faster than the MySQL version.

The relational database implementation has a slower response due to the more complicated DBMS system involved and increased overhead in programming because a greater number of programming statements must be written. However, the reason for using it is probably clear: the relational database provides a lot more flexibility in making queries, organizing the data, and, especially, expanding the application to handle a greater variety of data.

This flexibility is very important. The Rebase database from http://www.neb.com/rebase includes several more datafiles, such as where to get the enzymes. It wouldn't be difficult to add a table or tables to store that information in the relational version of my Perl program; it would be a major pain to keep adding new DBM hashes for various complex relationships. So, I want a relational database because it has good scalability as the database application grows.

6.8.1 RebaseDB Class: Accessing Restriction Enzyme Data

Following is the code for a new class, RebaseDB, which aims to provide the same functionality as the previous class Rebase, but with a relational database instead of a DBM hash data storage.

In the interest of space, the code for the following subroutines isn't reproduced here; look for them in Chapter 5: revcomIUB, complementIUB, and IUB_to_regexp. They are, of course, included in the RebaseDB.pm module available for downloading from this book's web page.

```
package RebaseDB;

#
# A simple class to provide access to restriction enzyme data from Rebase
#  including regular expression translations of recognition sites
# Data is stored in a MySQL database
#

use strict;
use warnings;
use DBI;
use Carp;

# Class data and methods
{
    # A hash of all attributes with default values
    my %_attributes = (
        rebase      => { },   # unused in this implementation
```

6.9 Additional Topics

In this short chapter, I have not mentioned several topics that are of considerable importance in database work:

- SQL has only been demonstrated, not fully laid out. It's a fairly simple language; if you'll be doing much database work, you should read the language manual that comes with your DBMS carefully, and look at several examples. Tutorial books are also available for most popular DBMS.
- Transactions are loosely defined as a set of database queries and modifications that belong together. For example, you may enter a new gene into your database by updating several relevant tables. If your system should experience a failure in the middle of such a set of updates, it can leave your database in an ill-defined state. By defining transactions, it's possible to avoid such undesirable states, and many DBMS now provide support for this view of database updates.
- Entity-relationship modeling is a top-down design methodology with a fairly simple graphics representation that signifies relationships between the data in a system.
- Stored procedures are parts of a database application that are performed at the point of entry when a user is filling out a form for instance. MySQL is just now starting to support them; major DBMS such as Oracle have had them for a long time. They tend to improve performance of an application when used well; they also greatly increase the difficulty of migrating to a different DBMS if that becomes necessary in the future.
- Object-oriented and object-relational databases are new data models that are finding some acceptance. You may come across them on the job, although their use is much more limited than the standard relational model.

6.10 Resources

The literature on databases is very large, and so the documentation for your particular RDMS is essential. For this book, I use MySQL, but many others are suitable.

Here are a handful of basic textbooks:

- Database Systems: A Practical Approach to Design, Implementation, and Management, Thomas Connolly and Carolyn Begg (Pearson Addison Wesley)
- A First Course in Database Systems, Jeffrey Ullman (Prentice Hall)
- An Introduction to Database Systems, C.J. Date (Pearson Addison Wesley)
- MySQL, Paul DuBois (SAMS)
- The MySQL Cookbook, Paul DuBois (O'Reilly & Associates)
- MySQL and Perl for the Web, Paul DuBois (SAMS)
- Programming the Perl DBI, Alligator Descartes and Tim Bunce (O'Reilly)

6.11 Exercises

Exercise 6.1

What's the difference between a database and a database management system? What's the difference between MySQL and Oracle?

Exercise 6.2

What are the limitations of the simple built-in Perl hash database DBM compared to a relational database? Its strengths?

Exercise 6.3

Is my homologs database, which was placed into second normal form, also in third normal form?

Exercise 6.4

Write a program that checks if a relation is in second normal form.

Exercise 6.5

RebaseDB.pm is written to specify a MySQL database. Rewrite the module so it can specify another relational database supported by DBI.

Exercise 6.6

The parse_rebase method of the RebaseDB.pm module uses several database queries per input line as part of the logic of avoiding duplicate entries for palindromes and reverse complements. Compare it with the parse_rebase method in Rebase.pm. Make a new parse_rebase method that works for both DBM and DBI database storage and will improve the efficiency of the DBI method by minimizing database queries.

Exercise 6.7

RebaseDB.pm is a port of the earlier version Rebase.pm that used the DBM hash database. Make a version of this module that handles both DBM and MySQL databases depending on the arguments passed to the new method.

Exercise 6.8

Compare MySQL with some other relational database management system; include such management issues as cost of purchase, cost of maintenance, availability of skilled personnel, stability of vendor in the marketplace, and customer support.

Exercise 6.9

Given the many possible alternate forms of a small set of simple relations, would you say that the relational model is not specific enough? What constraints might you add to improve the quality and reduce the number of possible relational designs?

Exercise 6.10

Name two bioinformatics problems that are ill-served by the data structures of a relational database.

Exercise 6.11

Implement a relational database that supports a project in your wet lab.

Chapter 7. Perl and the Web

One of the basic skills of a programmer is designing and putting up an interactive web site. This is as true for bioinformaticians as it is for other programmers.

Not every bioinformatician will need to implement a web site. Your work in bioinformatics may specialize early; perhaps you work in analyzing algorithms for representing gene cascades, while web programming is someone else's responsibility.

However, the Web has become the principal way to provide programs to users. This is certainly true in biology, where it is typical for laboratories to provide programs and access to data via the Web. Collaborations can be promoted between research groups, the need for publication of results can be addressed (witness the many peer-reviewed journal articles that invite readers to visit web sites for supporting information), and valuable research tools can be widely disseminated.

This chapter provides a quick introduction to the way web pages and the Internet work, followed by a closer look at some important parts of web programming. You'll see these techniques applied to create an interactive web page that accepts a sequence and enzyme names and returns a restriction map.

Along with the remarkable growth in use of the Web and the Internet have come an equal proliferation of languages, systems, and tools for web programming. There are now a variety of choices in how a programmer can create interactive web pages.

We will use one such system that employs the Perl language. To make the web pages interactive—to enable users to type in queries and get responses—we'll use the popular CGI.pm Perl module that's shipped with all recent versions of Perl. CGI stands for the Common Gateway Interface, a protocol that provides dynamic web content—web pages that change for various reasons (such as a user asking for a restriction map)—as opposed to static, unchanging web pages. Perl and CGI were an early success story in programming the Web and remain widely available as a standard web programming technique. They constitute a basic skill set in web programming, despite the many alternatives now available.

Because bioinformatics programmers need at least a basic working knowledge of web programming, I'll teach you the skills necessary to put an interactive web page on the Web; the example is based on a continuation of the restriction map example from previous chapters.

To begin, I'll give a brief run down of the chief components of web technology to give you the basic overview and terminology you'll need to do web programming and to read further in the field. If you're already familiar with servers, browsers, HTML, and the other components of web programming, feel free to skip ahead.

7.1 How the Web Works

The Internet is short for "interconnected networks." It is a set of conventions—protocols—with which computers and networks can intercommunicate. Its development from earlier work before 1980 allowed many different networks to join and users on many computers to communicate. This communication was originally done in several ways, such as by email, electronic mail, and by FTP, file transfer protocol. These methods remain very popular and widely used.

It wasn't until the early 1990s that the World Wide Web or Web was born as a new Internet service. The Web was based on the new hypertext transport protocol or HTTP, and the first software to use it was in the form of programs called web browsers and web servers. *Web browsers* are programs that handle user requests and display results to the user; the most widely known web browsers include Internet Explorer and Netscape. *Web servers* are programs that accept requests from web browsers and send results back to them for display; Apache is the most widely used web server. With the development of web browsers and their ability to handle images as well as text, these new protocols sparked intense popular interest in computers. At the same time, computer costs were falling steadily, and their capabilities were growing, which made the new web protocol even more widespread.

The Web has become critical to scientific programming; in fact, it started there. The Web and its associated protocols such as HTTP were originally developed at a high-energy physics laboratory in Switzerland, CERN, and they have been heavily used in the sciences ever since. In biology, as elsewhere, the Web has become one of the principal means of communication.

7.1.1 URLs

The Web is essentially a two-part system of browsers and servers, in which browsers get results from servers and display them for the user. This type of architecture is called a client-server design, in which the client (web browser) requests service from the server (web server). Web browsers and servers are just programs that run on computers. They may both be on the same computer, or, thanks to the Internet, they may be on opposite ends of the earth.

In order for this scheme to work, the web browser has to be able to send its request to the web server. For instance, say you want to see the New York Times from your Internet Explorer web browser (or your Netscape, Mozilla, or other web browser). You have to know the location of the New York Times on the Web and type it into the space provided in your browser screen.

So, you type http://www.nytimes.com, hit the Return or Enter key on your keyboard, and the next thing you know you're reading the latest articles about human cloning and double-stranded RNA. How does this work, exactly?

The answer is really very simple. The web browser sends your request to the Internet; the actual location of the desired computer is determined, and your request is sent to the web server program on that computer. The web server handles your request and sends back a web page your browser then displays. This web page may include other URLs of specific articles. You can click on one, and the whole process is repeated, but this time your request is for a specific article, which is then returned to your computer and displayed by your web browser.

Behind this simple overall architecture are several steps. A basic familiarity with some of these steps and the associated terminology is needed in order to learn the fundamentals of web programming.

The location you typed in, *http://www.nytimes.com*, is called a Uniform Resource Locator or URL. The Internet (to

7.2 Web Servers and Browsers

The whole architecture of the Web is based on the client-server, request-response protocol of HTTP. This colors everything that you do when you create interactive web pages.

Web browsers come in many flavors from many companies (or open source programming groups) and each has its own idiosyncrasies. Here, I'll stick to the basics, and the code should display okay on any standard web browser.

As you go forward, keep in mind the problem of incompatibility among web browsers and realize that at some point, it may become necessary for you to deal with the problem in the code you write for the Web.

7.3 The Common Gateway Interface

The preceding sections of this chapter have presented a brief overview of the design of the Web, the principal components of the programming environment such as HTTP, HTML, and URLs, and of the essential request-response nature of web interactions between web browsers and web servers. Now, it's time to look at CGI and a specific Perl module, CGI.pm, that is widely used to create interactive web pages on servers.

CGI, the Common Gateway Interface, is an interface between a web server and some other program that requests such web content as HTML documents or images.

A web browser may request from the web server the output from a CGI program. In this case, the web server finds the program or script, runs the program, and sends the output of the program back to the web browser. The output of the program may be HTML, just as it may be found in a static file, but it is created by the CGI script dynamically, so it may be different each time; for instance, it may include the time of day in its display.

In other words, a CGI script is just a program that produces web content that can be displayed in a web browser. It also can read information passed to it by the web server, usually parameters filled out by the user in a form displayed on a web browser that asks for the specifics of a query. For example, the parameters might be the name of a sequence file and the name of a restriction enzyme to map in the sequence. The CGI program takes the parameters, runs, and outputs a dynamically created web page to be returned to the user's web browser.

A CGI program can be written in just about any language, but the most common for CGI programs on the Web is Perl. Perl has a very nice module called CGI.pm that eases the task of writing CGI scripts; it's a popular way to create dynamic web sites with a minimum of bother.

7.3.1 Writing a CGI Program

So, how do you write a CGI Perl program? Basically, you write a Perl program that includes the line:

```
use CGI;
```

You then use the CGI.pm methods so that your Perl program outputs the code for the web page you want to return. After that, it's simply a matter of placing your new CGI script in the proper place in the web server's directory structure—namely, in a directory that the web server knows is supposed to contain CGI scripts. Finally, you type in the name of the CGI script to a web browser as a URL. The web browser sends the request to the web server, which executes the CGI script, collects its output, and returns it to your web browser to be displayed as a web page (or image, sound, or whatever).

You can actually write a Perl program that just prints out HTML code (without ever using the CGI.pm module) and install that as a CGI program. For instance, you can take the HTML page shown earlier and create a CGI Perl script that dynamically outputs that page. To prove it's dynamic, I'll add a little code that includes the time of day:

```
#!/usr/bin/perl

use strict;
use warnings;

my $time = localtime;

print "Content-type: text/html\n\n";
```

7.4 Rebase: Building Dynamic Web Pages

The simple examples in the previous sections showed how to load and use the CGI.pm module to display a very simple web page and how to examine the error logs of the web server to help debug a CGI program that doesn't display properly.

The real power of CGI comes from its ability to provide dynamic content—web pages that may display different information depending on such factors as when they're called, such as the date and time in the previous example. Dynamic content also handles the requests of users that are entered by typing in text fields, clicking on so-called "radio" buttons, selecting from lists, or other ways of inputting.

In this section, I'll show you how to use some of the modules from previous chapters, combined with the use of the CGI.pm module, to make an interactive, dynamic web page for displaying restriction maps. In this web page, the user will select which restriction enzyme or enzymes to search for and specify the sequence to search either by entering the sequence data into a text window or by browsing for the file that contains the sequence.

Here is the short CGI program, webrebase1, that accomplishes this. The main reason that it's short is because I've already developed modules for reading sequence files, for accessing the Rebase database, for calculating restriction maps, and for displaying the maps with simple text graphics. I can just reuse those modules here to accomplish my task:

```
#!/usr/bin/perl

# webrebase1 - a web interface to the Rebase modules

# To install in web, make a directory to hold your Perl modules in web space
use lib "/var/www/html/re";

use Restrictionmap;
use Rebase;
use SeqFileIO;
use CGI qw/:standard/;

use strict;
use warnings;

    print header,
    start_html('Restriction Maps on the Web'),
    h1('<font color=orange>Restriction Maps on the Web</font>'),
    hr,
    start_multipart_form,
    '<font color=blue>',
    h3("1) Restriction enzyme(s)?  "),
    textfield('enzyme'), p,
    h3("2) Sequence filename (fasta or raw format):  "),
    filefield(-name=>'fileseq',
        -default=>'starting value',
        -size=>50,
        -maxlength=>200,
    ), p,
    strong(em("or")),
    h3("Type sequence:  "),
    textarea(
        -name=>'typedseq',
        -rows=>10,
        -columns=>60,
        -maxlength=>1000,
    ), p,
```

7.5 Exercises

Exercise 7.1

Use your web browser to examine the actual HTML code for some of your favorite web pages. For example, on my Netscape browser, once a page is displayed, I can select the menu options View and then Page Source to see the HTML code.

Exercise 7.2

Identify the web server that is running on your computer. What version of the software is installed? Find the documentation for that web server. Are there good books available for that web server; free online tutorials, newsgroups, or FAQs; or local experts? Locate and examine the various logs for your web server. Where are the configuration files? How do you stop the web server, change a configuration file, and start the web server again? Is it a good idea to make a copy of a working configuration file before you change it?

Exercise 7.3

Is your computer located behind a firewall or a proxy server?

Exercise 7.4

Get the URL::URI module and read the documentation.

Exercise 7.5

Read the CGI module documentation. If available, look at a book about Perl and CGI.

Exercise 7.6

The sequence file uploaded by the CGI Perl script webrebase1 is assumed to be in FASTA format. Rewrite the program so that it uses the SeqFileIO.pm module and reports a problem back to the web page if that module can't determine the sequence file format.

Exercise 7.7

Write a new version of the CGI script webrebase1 that uses the RestrictDB.pm module instead of the Restrict.pm module; it will use a relational database store of Rebase information instead of a DBM hash-based store. Note that your DBMS will have to allow the web server to access its Rebase tables; this may occur as the user nobody or some other. Getting the permissions right is probably the hardest part of this exercise and is very much dependant on your operating system and web-server software.

Exercise 7.8

Write a web-based CGI Perl program that provides some information about your lab or an experiment that you've conducted. Include a means whereby users can send you email with questions.

Chapter 8. Perl and Graphics

The importance of an attractive and useful graphical display of information is fundamental to scientific programming. Just imagine the difference between seeing a graph of this week's stock market activity, and seeing a list of several pages of numbers describing the same activity. The former can be surveyed in a glance; the latter requires considerable study. The same principle applies to the display of results in bioinformatics. Of course, sometimes you need the raw data; but very often what you need is the overview that a good graphic can provide.

Although there are many programming systems to produce graphics, the most popular are those that can be displayed via web browsers. This chapter builds on what you learned in Chapter 7 by showing you how to add graphics to your display of scientific results on a web page.

My vehicle of choice for displaying scientific data is the GD.pm module, which allows you to produce images in the standard graphics formats PNG (Portable Network Graphics) and JPEG (Joint Pictures Expert Group), among others. I'll also briefly mention the graphics packages ImageMagick and Gimp, two more full-featured graphics creation and manipulation packages.

GD.pm is an interface to the gd graphics library written in the C programming language by Thomas Boutell. The gd graphics library is fairly basic; it's fast, it handles text and various kinds of lines and space filling by means of its various functions, and it is easy to program elements such as graphs.

The main reason for its success, however, based on those important technical reasons, is that it makes it possible to produce nice-looking graphics from your web site on-the-fly, in immediate response to a request from a user submitting a form.

8.1 Computer Graphics

Computer graphics programming is a specialty within computer science, and it has its own journals and conferences and laboratories. It encompasses just about everything you can see on a computer, from the fonts with which letters are drawn (both on the screen, and when computer typeset and printed like this book), to elaborate images, animation, and the special effects in movies.

Computer graphics images are represented as data in the computer; they're usually organized in files, one per image, although they can also be stored in a computer program's memory, as I did when storing the graphics for the Restrictionmap.pm module in a scalar variable. Some kinds of animation store several images in one file, but I won't cover those kinds of image files here.

For the purposes of this chapter, we won't be digging into the details of how graphics files are designed; that level of detail isn't possible in a single chapter. Here, I'll just discuss the very basic information you need to get started generating and displaying images for your web site. I'm going to show you some easy (and inexpensive) ways to generate graphics, enough to set your feet on the right path. The modules I use have methods that handle the details of reading and writing the different file formats, so you can simply use the methods and ignore the details of the formats themselves.

8.1.1 Basic Graphics Concepts

There are two things you need for computer graphics: a program to create a graphics file and the correct software and hardware to display the image. These days, digital photography may be the source of digital images, or a printed image may be entered into digital format by a scanning device. As computer technology has developed, many graphics display devices and file formats have seen some use and then virtually disappeared as they are supplanted by other systems.[1] However, even though there are many file formats that are currently in use, a few standard formats are widely known and can be displayed by most graphics display programs. The PNG and JPEG files I use here are very commonly known.

[1] One of my first programming jobs was to write vector graphics for a Tektronix oscilloscope display. Vector graphics are specified by endpoints of lines. Although such graphic programs are still in use, they are almost always converted to raster graphics for display these days. The same year, I wrote a large graphics program to write and play computer music on a Blit terminal—a raster graphics display that was the first to have modern-style simultaneously updated windows. Both systems have faded away, but the software they were written in was ported to other display devices.

Although I won't go into any detail about graphics file formats, there are a few basic facts you need to understand because you'll be asked to specify some of these parameters when you use a graphics library (such as GD.pm). For instance, in the Perl code that follows, you'll see a method colorAllocate that specifies and enters a color into a color table. The few short comments that follow in this section are meant to introduce these, and other common terms, and to give brief definitions.

These days the standard graphics formats are mostly some variation of a raster image, in which a rectangular image is described by a two-dimensional matrix of pixel values. A pixel (short for "picture element") is one glowing dot on your computer screen. These values can be represented in different ways, from a simple 1 or 0 representing black or white (called a bitmap), to a number representing various shades of grey, to various color image schemes. A computer display may have different numbers of pixels overall; for instance, the one I'm working on now has 1024 horizontal and 768 vertical rows. Displays (or printers) are also rated by how closely packed the pixels are, usually by saying they have so many DPI or dots per inch.

8.2 GD

GD is an excellent, relatively simple graphics toolkit for creating or modifying graphic images. You may create new images or read in images from several formats to edit or merge. GD contains a number of graphics *primitives*, objects such as lines and rectangles, which can be added to an image. It also includes support for TrueType fonts.

8.2.1 Installing GD

To install GD.pm on your computer, you should first test to see if the module is already there: try typing perldoc GD to see if the documentation that is installed with the module is running on your computer. If not, GD.pm is available from CPAN, which is the best way to get it (its real home is at http://stein.cshl.org).

The GD documentation explains which supporting libraries are necessary. In particular, you need to have Thomas Boutell's gd C library installed, and in the proper version, to work with the version of GD.pm you are installing. To work with different graphics file formats, the gd library is best compiled in the presence of additional libraries that handle PNG, JPEG, zlib compression, and the FreeType version of TrueType fonts. How to obtain and install these additional packages is all explained in the GD documentation. You may need extra time to get these various pieces into place before you can get GD to do everything it can do.

So, even if you use CPAN to install GD, the installation requires some information about those additional packages; it's a good idea to look at the documentation first, and to get those libraries in place, before installing GD. To install on my Linux system, I become root and issue the following commands:

```
perl -MCPAN -e shell;
cpan> install GD
```

8.2.2 Using GD

The following list is a partial overview of GD's capabilities:

- To create a new image, you call the new constructor with x and y values of the size in pixels of the desired image. You can also choose between a palette or a truecolor image. And you can either create a new image or open an existing image in a few common formats (such as PNG, JPEG, GD, and XPM).
- You can output images as PNG, JPEG, WBMP (a bitmap image format), or its own GD format and the compressed version GD2.
- The color table can be manipulated in several ways. A new color can be added or an old one deleted. The closest match in the color table to specified red, green, and blue values can be returned, and the RGB values of an existing color table entry can be determined. One color can be designated as transparent so that portions of an image drawn in that color are invisible.
- Lines can be drawn using defined "brushes," or in certain defined styles (like dotted lines, for example). Shapes may be filled with repeated copies of another image, or "tiled."
- You can draw individual pixels, lines, dashed lines, rectangles, polygons, and arcs. You can fill regions of the

8.3 Adding GD Graphics to Restrictionmap.pm

Now that you've seen an overview of the GD.pm Perl module, let's enhance the Restrictionmap.pm module that you've seen previously in Chapter 5, Chapter 6, and Chapter 7 so it can output PNG graphics files.

Reviewing how the Restrictionmap.pm module works, you can see that its get_graphic method examines the _graphictype attribute to determine what graphics drawing method to call. Following is the get_graphic method again, along with the attributes the class defines. (Recall that this class inherits from the class Restriction.pm, so many things are defined there):

```
 # Class data and methods
{
    # A list of all attributes with default values.
    my %_attributes = (
        #    key   = restriction enzyme name
        #    value = space-separated string of recognition sites => regular expressions
        _rebase      => { },    # A Rebase.pm object
        _sequence    => '',     # DNA sequence data in raw format (only bases)
        _enzyme      => '',     # space separated string of one or more enzyme names
        _map         => { },    # a hash: enzyme names => arrays of locations
        _graphictype => 'text', # one of 'text' or 'png' or some other
        _graphic     => '',     # a graphic display of the restriction map
    );

    # Return a list of all attributes
    sub _all_attributes {
            keys %_attributes;
    }
}

sub get_graphic {
    my($self) = @_;

    # If the graphic is not stored, calculate and store it
    unless($self->{_graphic}) {

        unless($self->{_graphictype}) {
            croak 'Attribute graphictype not set (default is "text")';
        }

        # if graphictype is "xyz", method that makes the graphic is "_drawmap_xyz"
        my $drawmapfunctionname = "_drawmap_" . $self->{_graphictype};

        # Calculate and store the graphic
        $self->{_graphic} = $self->$drawmapfunctionname;
    }

    # Return the stored graphic
    return $self->{_graphic};
}
```

As you recall, the get_graphic method requires that some graphic type be defined in the _graphictype attribute. It then tries to call a method whose name incorporates the name of the graphic type; for instance, if the graphic type is "png", the get_graphic method tries to call a graphic drawing method called _drawmap_png.

How might you design such a get_graphic method?

8.3.1 Designing Graphics

8.4 Making Graphs

Displaying data graphically (graphs, bar graphs, pie charts, etc.) is the most common goal of image programming You've just seen how the GD Perl module helps you create graphics. It's great for quickly providing images from CGI web programs.

GD::Graph, another Perl module that uses GD as its underlying mechanism, does an excellent job of rendering all sorts of graphs. Programming graphs with GD::Graph is relatively easy, especially if you start with the example programs and modify them for your own purposes. The module is object oriented, and it provides many options for displaying the various graphs. Plan to take some time exploring the documentation and trying different options as you begin programming with GD::Graph. Its flexible yet relatively simple programming interface has made it a popular and powerful module. GD::Graph can be found on CPAN. It's simple to install once you have GD itself installed.

For more in-depth information on graphics programming in Perl, see Perl Graphics Programming by Shawn Wallace (O'Reilly). The book includes a chapter on GD::Graph that can serve as a fine tutorial. The documentation for GD::Graph (on CPAN, or type perldoc GD::Graph in a terminal window) is well-written and clearly organized. The software comes with several example programs that will help get you started.

The following short program, gd1.pl, uses GD::Graph to create a simple bar graph.

```
#!/usr/bin/perl

#
# gd1.pl -- create GD::Graph bar graph
#

use strict;
use warnings;
use Carp;
use GD::Graph::bars;

my %dataset = ( 1 => 3,
                2 => 17,
        3 => 34,
        4 => 23,
        5 => 25,
        6 => 20,
        7 => 12,
        8 => 3,
        9 => 1
);

# create new image
my $graph = new GD::Graph::bars(600, 300);

# discover maximum values of x and y for graph parameters
my( $xmax) = sort {$b <=> $a} keys %dataset;
my( $ymax) = sort {$b <=> $a} values %dataset;
# how many ticks to put on y axis
my $yticks = int($ymax / 5) + 1;

# define input arrays and enter 0 if undefined x value
my(@xsizes) = (0 .. $xmax);
my(@ycounts) = ();
foreach my $x (@xsizes) {
    if ( defined $dataset{$x}) {
        push @ycounts, $dataset{$x};
    }else{
```

8.5 Resources

I recommend the book The Visual Display of Quantitative Information by Edward Tufte (Graphics Press).

Although I don't demonstrate their use in this book, I want to mention two more full-featured graphics software packages that are free, and very useful if you'll be doing graphics to any extent:

- ImageMagick and its Perl interface module Image::Magick is a valuable tool, especially for creating images. Over 60 different file formats are supported, compared to the much more limited number in GD. A wide range of image-manipulation techniques are supported, enabling you to create animation, thumbnails, and apply many image processing functions, for example.
- The Gimp (GNU Image Manipulation Program) is very much like the Photoshop program. It has an interface to Perl scripts that is built-in, and there are several modules, starting with the Gimp.pm module, that provide Perl programmers with access to its functionality.

Between GD, Gimp, and ImageMagick, the Perl programmer has ready access to a host of image creation and manipulation software, at no cost.

8.6 Exercises

Exercise 8.1

Add an argument "linelength=>50" to Restrictionmap to set the length of lines in the graphic output.

Exercise 8.2

What happens if you request a graphic type for which no corresponding method has been defined? Would there be a better way to handle this situation than the current behavior of the module?

Exercise 8.3

If more than one enzyme is present, draw each enzyme in a different color. You'll have to solve the problem of how to define the required number of colors and how to choose them so that they'll be easily distinguishable from each other. You'll also have to rework the annotation lines when more than one type of enzyme appears on an annotation line, drawing the different sections individually with different colors.

Exercise 8.4

Add header information such as filename, count of restriction sites for each enzyme, and date, to the text or PNG or JPEG graphic generated by _drawmap_text or _drawmap_png or _drawmap_jpg.

Exercise 8.5

In _drawmap_text, add a number at the beginning of each line of sequence giving the position of the first base on that line. First figure out the number of digits needed for the last line, so you can make the appropriate amount of space before each line. Can you right-justify the numbers in the allotted space?

Exercise 8.6

In _drawmap_text, put a space between each group of 10 bases on a line to make it easier to find a particular base position.

Exercise 8.7

In _drawmap_png or _drawmap_jpg, instead of putting the enzyme names in annotation lines above the sequence lines, try simply printing the sequence lines, but highlight each restriction site with color and enclosing it in a box. Try giving each type of enzyme in the map a different color and type of box and add a key that shows which enzyme matches up with each color/box.

This works fine with a single enzyme. Does it work with two enzymes? With three or more? Do you need to change the output of _drawmap_text to accomplish this?

Exercise 8.8

Rewrite the webrebase1 CGI Perl script so that it is two programs: one that displays the opening screen form, and the other that calculates the PNG image. Say you call the image-generating CGI script rebase_png. Then use the CGI for an HTML tag such as:

```
print img({ -src => "/cgi-bin/rebase_png?enzyme=EcoRI;fileseq=
    sampleecori.dna2" });
```

In this way, you can embed a dynamically generated image into a larger HTML document; it's an alternative to the method shown in the text in which the image is shown by itself in the web browser.

Exercise 8.9

The _drawmap_jpg and _drawmap_png methods are almost identical. Add a third method that does most of the work of these methods, taking as an argument the file type desired for output. Then rewrite _drawmap_jpg and _drawmap_png so that they're much shorter. Now add a _drawmap_wbmp method.

Exercise 8.10

The _drawmap_png method assumes that annotation lines contain spaces while sequence lines do not. This is called

Chapter 9. Introduction to Bioperl

Bioperl is a collection of more than 500 Perl modules for bioinformatics that have been written and maintained by an international group of volunteers. Bioperl is free (under a very unrestrictive copyright), and its home is http://bioperl.org.

One of the most difficult things about Bioperl is getting started using it. This is due to a scarcity of good documentation (which is being rectified) as well as the sheer size of the Bioperl module library. This chapter will help you get started using the Bioperl project software; it will guide you through the initial steps of getting the software, installing it, and exploring the tutorial and example material that it provides. After working through this chapter, you'll be well prepared to delve deeper into the riches of Bioperl, and, if you've also worked through the object-oriented chapters earlier in this book, you'll be in a good position to read the Bioperl code and contribute to the project yourself.

The modules in Bioperl are written in the object-oriented style. Perl programmers who do not know object-oriented programming can still use the Bioperl modules with just a bit of extra information, as outlined in Chapter 3.

The Bioperl modules cover various areas of bioinformatics, including some you've seen previously in this book. Although Bioperl includes some example programs, it is not meant to be a collection of complete user-ready programs. Rather, it's implemented as a toolkit you can dip into for help when writing your own programs. Its goal is to provide good working solutions to common bioinformatics tasks and to speed your program development.

One of the best things about Bioperl is that it's an open source project, meaning that interested developers are invited to contribute by writing code or in other ways, and the code is available to anyone interested. If you've learned enough about Perl for bioinformatics to have worked through a good portion of this book, you'll find plenty of opportunity to get involved in Bioperl if you have the time and inclination.

9.1 The Growth of Bioperl

The practice of freely distributing Perl software for bioinformatics over the Internet began around 1992. Gradually, Perl became more and more popular for biology applications.[1] The release of Perl Version 5 and its support for object-oriented programming accelerated the development of reusable modules for biology at many research centers around the world. The large-scale genome sequencing efforts, then underway, provided much of the impetus, as well as the talent and funding, for these efforts. The Bioperl project, officially organized in 1995, coalesced around one of these bioinformatics research groups that was doing a good job of organizing the collective volunteer effort such collaborative projects require. More information, including a short history and list of contributors, can be found at the Bioperl site http://bioperl.org. The Bioperl web site is frequently being updated and improved, and is the primary source of Bioperl code and documentation.

[1] See, for example, the article "How Perl Saved the Human Genome Project" by Lincoln Stein at http://bioperl.org/GetStarted/tpj_ls_bio.html.

Today, the Bioperl project has grown to a point where it is both useful enough, and well enough documented, that it is a must for Perl programming in bioinformatics. The documentation includes a program bptutorial.pl that comes with Bioperl, which explains and demonstrates several areas of the project (more on that later in this chapter).

9.2 Installing Bioperl

Installing Bioperl's large collection of modules isn't too difficult. It usually goes fairly painlessly, even though there are a few extra installation steps due to the additional outside programs and Perl modules on which Bioperl depends. You will probably want to use Perl's CPAN to fetch and install Bioperl.

INSTALL is a good document that covers Bioperl installation on Unix/Linux, Windows, and Mac operating systems. It's part of the Bioperl distribution and is located at http://bioperl.org/Core/Latest/INSTALL. This location may change, but it's easy to find from the Download link on the main Bioperl web page. If you're going to install Bioperl, I recommend you read this document first; here, I'll give an overview of what's available and add a few additional comments that may help with installation.

Here's how to get Bioperl on different platforms:

- On Unix/Linux, if you download the *tar* file from the web site, you merely need to untar it and go through the configure and make process by hand, as described in the INSTALL file that comes with the distribution.
- If you have a Microsoft Windows machine with ActiveState's Perl (http://www.activestate.com), there's a PPM file available for Bioperl; at the time of this writing, it's at http://bioperl.org/ftp/DIST/Bioperl-1.2.1.ppd.
- There is also a CVS repository for Bioperl from which you can fetch the most current versions of the modules. But be careful: some newer versions of the modules are implementing new features and have more bugs than are found in some of the older, more stable releases. The details of how to install from CVS are also available at the Bioperl web page.

All these methods for installing Bioperl are fine, but probably the most common way for Perl programmers to install sets of modules is by way of CPAN. I discussed CPAN in Chapter 1, but it's worth discussing again as it relates to Bioperl.

To install Bioperl, you start by typing the following at the command line:

```
perl -MCPAN -e shell;
```

This gives the CPAN shell prompt:

```
cpan>
```

It's often the case that a module you want to install may require other modules for its proper operation, and perhaps one or more of these additional modules have not yet been installed. The CPAN system includes a way to check to see what other modules are required, and you can configure it to automatically follow and install missing prerequisites.

Especially with a large collection of modules like Bioperl, you may expect to see such prerequisites crop up. When I asked my CPAN session how it was configured:

```
cpan> o conf
```

one of the lines of output from that query was:

```
prerequisites_policy ask
```

9.3 Testing Bioperl

To check that things were working okay with my new Bioperl installation, I first wrote a little test to see if a Perl program could find the Bio::Perl module:

```
use Bio::Perl;
```

I ran it by putting it in a file bp0.pl and giving it to the Perl interpreter:

```
[tisdall]# perl bp0.pl
[tisdall]#
```

As you see, it didn't complain, which means Perl found the Bio::Perl module. If it can't find it, it will complain. When I copied the bp0.pl program to a file called bp0.pl.broken and changed the module call of Bio::Perl to a call for the nonexistent module Bio::Perrl, I got the following (slightly truncated) error output:

```
[tisdall@coltrane development]$ perl bp0.pl.broken
Can't locate Bio/Perrl.pm in @INC
BEGIN failed--compilation aborted at bp0.pl.broken line 1.
```

9.3.1 Second Test

Now I knew that Bio::Perl could be found and loaded. I next tried a couple of test programs that are given in the bptutorial.pl document. I found a link to that tutorial document on the Web from the http://bioperl.org page. Alternately, I could have opened a window and typed at the command prompt:

```
perldoc bptutorial.pl
```

I went to the section near the beginning of the document called "I.2 Quick getting started scripts" and created a file tut1.pl on my computer by pasting in the text of the first tutorial script:

```
   use Bio::Perl;

  # this script will only work with an internet connection
  # on the computer it is run on
  $seq_object = get_sequence('swissprot',"ROA1_HUMAN");

  write_sequence(">roa1.fasta",'fasta',$seq_object);
```

I then ran the program and looked at the output file:

```
[tisdall]$ perl tut1.pl
[tisdall]$ cat roa1.fasta
>ROA1_HUMAN Heterogeneous nuclear ribonucleoprotein A1 (Helix-destabilizing protein)
(Single-strand binding protein) (hnRNP core protein A1).
SKSESPKEPEQLRKLFIGGLSFETTDESLRSHFEQWGTLTDCVVMRDPNTKRSRGFGFVT
YATVEEVDAAMNARPHKVDGRVVEPKRAVSREDSQRPGAHLTVKKIFVGGIKEDTEEHHL
RDYFEQYGKIEVIEIMTDRGSGKKRGFAFVTFDDHDSVDKIVIQKYHTVNGHNCEVRKAL
SKQEMASASSSQRGRSGSGNFGGGRGGGFGGNDNFGRGGNFSGRGGFGGSRGGGGYGGSG
DGYNGFGNDGGYGGGGPGYSGGSRGYGSGGQGYGNQGSGYGGSGSYDSYNNGGGRGFGGG
SGSNFGGGGSYNDFGNYNNQSSNFGPMKGGNFGGRSSGPYGGGGQYFAKPRNQGGYGGSS
SSSSYGSGRRF
[tisdall]$
```

That seemed to work perfectly.

9.3.2 Third Test

I tried the next short script from the same section of the tutorial, pasting it into a file called tut2.pl:

```
[tisdall]$ cat tut2.pl
use Bio::Perl;
```

9.4 Bioperl Problems

Bioperl is still a work in progress, and it has some problems. I'd like to mention the two main problems now.

First, the Bioperl documentation is incomplete. In fact, until fairly recently, there was no document that provided a tutorial introduction to the project. This has changed; the bptutorial.pl document, which you've already seen and will see more of, is an excellent beginning, despite its occasional errors. This document cleverly combines a tutorial with quite a few example programs that you can run, as you'll soon see.

Other documentation for Bioperl is also available, including Internet-based tutorials, forthcoming books, example programs, and journal articles. So, the situation has recently improved.

Second, Bioperl is big (over 500 modules), written by volunteers, and gradually evolving. The size of the project is a sign that Bioperl addresses many interesting and useful problems, but it also means that, for the new user of Bioperl, an overview of the available resources is a task in itself.

The majority of the Bioperl code is quite good, especially the most-used parts of it. However, the volunteer and evolving nature of Bioperl development means that some of the code is unfinished and not as well integrated with other parts of the project as one would like. Newer or less used modules may still need some shaking out by users in real-world situations. This is where you can make an initial contribution to the project: as you find problems, report them (more on that later).

Many of the computing world's most successful programs are the result of the same kind of volunteer development as Bioperl (the Perl language itself and the Apache web server are two examples). Bioperl is well positioned to achieve a similarly central position in the field of bioinformatics.

9.5 Overview of Objects

Bioperl is a big project and a fairly large collection of modules. Some of these modules are standalone; others interact with each other in various ways.

Your first task in learning about Bioperl is to get an idea of the main subject areas the modules are designed to address. So to begin with, here is a brief overview of the main types of objects in Bioperl, collected in a few broadly defined groups.

Sequences Bio::Seq is the main sequence object in Bioperl.Bio::PrimarySeq is a sequence object without features.Bio::SeqIO provides sequence file input and output.Bio::Tools::SeqStats provides statistics on a sequence.Bio::LiveSeq::* handles changing sequences.Bio::Seq::LargeSeq provides support for very large sequences. *Databases* Bio::DB::GenBank provides GenBank access. Similar modules are available for several biological databases.Bio::Index::* indexing and accessing local databases.Bio::Tools::Run::StandAloneBlast runs BLAST on your local computer.Bio::Tools::Run::RemoteBlast runs BLAST remotely.Bio::Tools::BPlite parses BLAST reports.Bio::Tools::BPpsilite parses psiblast reports.Bio::Tools::HMMER::Results parses HMMER hidden Markov model results. *Alignments* Bio::SimpleAlign manipulates and displays simple multiple sequence alignments.Bio::UnivAln manipulates and displays multiple sequence alignments.Bio::LocatableSeq are sequence objects with start and end points for locating relative to other sequences or alignments.Bio::Tools::pSW aligns two sequences with the Smith-Waterman algorithm.Bio::Tools::BPbl2seq is a lightweight BLAST parser for pairwise sequence alignment using the BLAST algorithm.Bio::AlignIO also aligns two sequences with BLAST.Bio::Clustalw is an interface to the Clustalw multiple sequence alignment package.Bio::TCoffee is an interface to the TCoffee multiple sequence alignment package.Bio::Variation::Allele handles sets of alleles.Bio::Variation::SeqDiff handles sets of mutations and variants. *Features and genes on sequences* Bio::SeqFeature is the sequence feature object in Bioperl.Bio::Tools::RestrictionEnzyme locates restriction sites in sequence.Bio::Tools::Sigcleave finds amino acid cleavage sites.Bio::Tools::OddCodes rewrites amino acid sequences in abbreviated codes for specific statistical analysis (e.g., a hydrophobic/hydrophilic two-letter alphabet).Bio::Tools::SeqPattern provides support for regular expression descriptions of sequence patterns.Bio::LocationI provides an interface to location information for a sequence.Bio::Location::Simple handles simple location information for a sequence, both as a single location and as a range.Bio::Location::Split provides location information where the location may encompass multiple ranges, and even multiple sequences.Bio::Location::Fuzzy provides location information that may be inexact.Bio::Tools::Genscan is an interface to the gene finding program.Bio::Tools::Sim4::Results (and Exon) is an interface to the gene exon finding program.Bio::Tools::ESTScan is an interface to the gene finding program.Bio::Tools::MZEF is an interface to the gene finding program.Bio::Tools::Grail is an interface to the gene finding program.Bio::Tools::Genemark is an interface to the gene finding program.Bio::Tools::EPCR parses the output of ePCR program.

9.6 bptutorial.pl

I've already shown you a little of the bptutorial.pl document. I ran and discussed a few of the short example programs in the preceding sections.

As you know, one of the easiest ways to get started with a programming system is to find some working and fairly generic programs in that system. You can read and run the programs, and then proceed to alter them using them as templates for your own programming development.

Bioperl comes with a directory of example programs, but the best place to begin looking for starting-off program code is right in the bptutorial.pl document itself. That .pl suffix on the name is the giveaway; the document is actually itself a program, cleverly designed so that you can read, and run, example programs that exercise the core parts of the Bioperl project.

The following explanation of the runnable programs that are part of bptutorial.pl appears at the end of the document (when you view it on the Web or as the output of perldoc bptutorial.pl).

```
 V.2 Appendix: Tutorial demo scripts

The following scripts demonstrate many of the features of
bioperl. To run all the core demos, run:

 > perl -w  bptutorial.pl 0

To run a subset of the scripts do

 > perl -w  bptutorial.pl

and use the displayed help screen.

It may be best to start by just running one or two demos
at a time. For example, to run the basic sequence manipu-
lation demo, do:

 > perl -w  bptutorial.pl 1

Some of the later demos require that you have an internet
connection and/or that you have an auxilliary bioperl
library and/or external cpan module and/or external pro-
gram installed.  They may also fail if you are not running
under Linux or Unix.  In all of these cases, the script
should fail "gracefully" simply saying the demo is being
skipped.  However if the script "crashes", simply run the
other demos individually (and perhaps send an email to
bioperl-l@bioperl.org detailing the problem :-).
```

(Recall that the -w flag to Perl turns on warnings in almost the same manner as a use warnings; directive.)

To test my Bioperl installation, I started by running the basic sequence manipulation demo as suggested.

First, I thought I might copy the bptutorial.pl program file into my own working directory from the Bioperl distribution directory where I'd unpacked the source code. I wanted to put it in my own directory so as not to muddy up the Bioperl distribution directory with my own extraneous files. However, I discovered that the tutorial demo programs rely on a number of datafiles that are found in the *t/data/* subdirectory of the Bioperl distribution. Running the

9.7 bptutorial.pl: sequence_manipulation Demo

In this section, I'll go through the code for the demo subroutine sequence_manipulation that was shown in the last section.

The subroutine is actually an anonymous subroutine; a reference to the subroutine is saved in the scalar reference variable $sequence_manipulation:

```
$sequence_manipulations = sub {
...
}
```

The first few lines of code declare some variables with my. Notice that these are not being passed in as arguments; this method uses no arguments but does occasionally use global variables such as $dna_seq_file, which, as you've just seen, contain the pathname of the input sequence file the demo will use:

```
my ($infile, $in, $out, $seqobj);
$infile = $dna_seq_file;

print "\nBeginning sequence_manipulations and SeqIO example... \n";
```

The code is cross-referenced to the tutorial sections of the file. The next comment line refers to the part of the document:

```
# III.3.1 Transforming sequence files (SeqIO)
```

which can be looked up in the table of contents to the document for further reading:

```
III.3 Manipulating sequences
III.3.1 Manipulating sequence data with Seq methods (Seq)
```

Now, I'll take a look at the first section of example code in the sequence_manipulations method:

```
# III.3.1 Transforming sequence files (SeqIO)

$in  = Bio::SeqIO->new('-file' => $infile ,'-format' => 'Fasta');
$seqobj = $in->next_seq(  );

# perl "tied filehandle" syntax is available to SeqIO,
# allowing you to use the standard <> and print operations
# to read and write sequence objects, eg:
#$out = Bio::SeqIO->newFh('-format' => 'EMBL');

$out = Bio::SeqIO->newFh('-format' => 'fasta');

print "First sequence in fasta format... \n";
print $out $seqobj;
```

The code starts with a call to the new object constructor of the Bio::SeqIO class. The new method is being passed the pathname to a FASTA file in $infile, and told that the format is FASTA.

A quick look at the Bio::SeqIO documentation explains that the call to Bio::SeqIO->new returns a stream object for the specified format. So, $out is a stream object (a stream is input or output of data) for FASTA-formatted data, and $in is a stream object for FASTA-formatted input from the file named in the $infile variable. These $in and $out objects are also filehandles.

After the $in object is initialized on the FASTA file named in $infile, it calls the next_seq method, which gets the next (in this case, the first and perhaps only) FASTA record from the file, and it creates a sequence object $seqobj. The output $out object is created. The Perl print statement is then called, using $out as a filehandle, and printing $seqobj

9.8 Using Bioperl Modules

I've reached my stated goal: to get you started using Bioperl. Needless to say, there is a great deal more to explore than will fit into the confines of this chapter.

For those who wish to continue, here is a short list of some of the interesting and useful parts of Bioperl that will repay your efforts to learn them with considerably increased programming power:

- Overview of Bioperl objects
- Seq objects
- Gbrowse and gff files
- BLAST parsing
- Automated database searching

Part III: Appendixes

Appendix A. Perl Summary

This appendix summarizes those parts of the Perl programming language that will be most useful to you as you read this book. It is not a comprehensive summary of the Perl language. Remember that Perl is designed so that you don't need to know everything in order to use it. Source material for this appendix came from Programming Perl (O'Reilly).

A.1 Command Interpretation

The Perl programs in this book start with a line something like:

```
#!/usr/bin/perl
```

On Unix (or Linux) systems, the first line of a file can include the name of a program and some flags, which are optional. The line must start with #!, followed by the full pathname of the program (in this case, the Perl interpreter), followed optionally by a single group of one or more flags. It's common in Perl programs to see the -w flag on this first command interpreter line, like so:

```
#!/usr/bin/perl -w
```

The -w flag turns on extra warnings. I prefer to do that with the line:

```
use warnings;
```

because it's more portable to different operating systems.

If the Perl program file is called myprogram and has executable permissions, you can type myprogram (or possibly ./myprogram or the full or relative pathname for the program) to start the program running.

The Unix operating system starts the program specified in the command interpretation line and gives it as input the rest of the file after the first line. So, in this case, it starts the Perl interpreter and gives it the program in the file to run.

This is just a shortcut for typing the following at the command line:

```
/usr/bin/perl myprogram
```

A.2 Comments

A comment begins with a # sign and continues from there to the end of the same line. It is ignored by the Perl interpreter and is only there for programmers to read. A comment can include any text.

A.3 Scalar Values and Scalar Variables

A scalar value is a single item of data, such as a string, a number, or a reference.

A.3.1 Strings

Strings are scalar values and are written as text enclosed within single quotes, like so:

```
'This is a string in single quotes.'
```

or double quotes, such as:

```
"This is a string in double quotes."
```

A single-quoted string prints out exactly as written. With double quotes, you can include a variable in the string, and its value will be inserted or "interpolated." You can also include commands such as \n to represent a newline (see Table A-3):

```
$aside = '(or so they say)';
$declaration = "Misery\n $aside \nloves company.";
print $declaration;
```

This snippet prints out:

```
Misery
 (or so they say)
loves company.
```

A.3.2 Numbers

Numbers are scalar values that can be:

- Integers:

```
 3
-4
 0
```

- Floating-point (decimal):

```
4.5326
```

- Scientific (exponential) notation (3.13 x 1023 or 313000000000000000000000):

```
3.13E23
```

- Hexadecimal (base 16):

```
0x12bc3
```

- Octal (base 8):

```
05777
```

- Binary (base 2):

```
0b10101011
```

A.4 Assignment

Scalar variables are assigned scalar values with an assignment operator (the equals sign) in an assignment statement:

```
$thousand = 1000;
```

assigns the integer 1000, a scalar value, to the scalar variable $thousand.

The assignment statement looks like an equal sign from elementary mathematics, but its meaning is different. The assignment statement is an instruction, not an assertion. It doesn't mean "$thousand equals 1000." It means "store the scalar value 1000 into the scalar variable $thousand". However, after the statement, the value of the scalar variable $thousand is, indeed, equal to 1000.

References are usually saved in scalar variables. For example:

```
$pi = \3.14159265;
```

If you try to print $pi after this assignment, you get an indication that it's a reference to a scalar value at a memory location represented in hexadecimal digits. To print the value of a variable that's a reference to a scalar, precede its name with an additional dollar sign:

```
print $pi,"\n";
print $$pi, "\n";
```

This gives the output:

```
SCALAR(0x811d1bc)
3.14159265
```

You can assign values to several scalar variables by surrounding variables and values in parentheses and separating them by commas, thus making lists:

```
($one, $two, $three) = ( 1, 2, 3);
```

There are several assignment operators besides = that are shorthand for longer expressions. For instance, $a += $b is equivalent to $a = $a + $b. Table A-1 is a complete list.

Table A-1. Assignment operator shorthands

Example of operator	**Equivalent**	
$a += $b	$a = $a + $b	(addition)
$a -= $b	$a = $a - $b	(subtraction)
$a *= $b	$a = $a * $b	(multiplication)
$a /= $b	$a = $a / $b	(division)

A.5 Statements and Blocks

Programs are composed of statements often grouped together into blocks.

A statement ends with a semicolon (;), which is optional for the last statement in a block.

A block is one or more statements usually surrounded by curly braces:

```
{
  $thousand = 1000;
  print $thousand;
}
```

Blocks may stand by themselves but are often associated with such constructs as loops or if statements.

A.6 Arrays

Arrays are ordered collections of zero or more scalar values, indexed by position. An array variable begins with the @ sign followed by a legal variable name. For instance, here are two possible array variable names:

```
@array1
@dna_fragments
```

You can assign scalar values to an array by placing the scalar values in a list separated by commas and surrounded by a pair of parentheses. For instance, you can assign an array the empty list:

```
@array = (  );
```

or one or more scalar values:

```
@dna_fragments = ('ACGT', $fragment2, 'GGCGGA');
```

Notice that it's okay to specify a scalar variable such as $fragment2 in a list. Its current value, not the variable name, is placed into the array.

The individual scalar values of an array (the elements) are indexed by their position in the array. The index numbers begin at 0. You can specify the individual elements of an array by preceding the array name by a $ and following it with the index number of the element within square brackets, like so:

```
$dna_fragments[2]
```

This equals the value of 'GGCGGA', given the values previously set for this array. Notice that the array has three scalar values indexed by numbers 0, 1, and 2. The third and last element is indexed 2, one less than the total number of elements 3, because the first element is indexed number 0.

You can make a copy of an array using an assignment operator =, as in this example that makes a copy @output of an existing array @input:

```
@output = @input;
```

If you evaluate an array in scalar context, the value is the number of elements in the array. So if array @input has five elements, the following example assigns the value 5 to $count:

```
$count = @input;
```

Figure A-1 shows an array @myarray with three elements, which demonstrates the ordered nature of an array by which each element appears and can be found by its position in the array.

Figure A-1. Schematic of an array

You can make a reference to an array by preceding it with a backslash; you dereference it by preceding the reference with an at sign @ for the entire array or with an extra dollar sign $ for an individual element of the array:

```
@a = ( 'one', 'two', 'three');
$aref =  a;
print $aref, "\n";
print $$aref[0], "\n";
print "@$aref", "\n";
```

gives the output:

A.7 Hashes

A hash (also called an associative array) is a collection of zero or more pairs of scalar values, called keys and values. The values are indexed by the keys. An array variable begins with the % sign followed by a legal variable name. For instance, possible hash variable names are:

```
%hash1
%genes_by_name
```

You can assign a value to a key with a simple assignment statement. For example, say you have a hash called %baseball_stadiums and a key Phillies to which you want to assign the value Citizens Bank Park. This statement accomplishes the assignment:

```
$baseball_stadiums{'Phillies'} = 'Citizens State Bank';
```

Note that a single hash value is referenced by a $ instead of a % at the beginning of the hash name; this is similar to the way you reference individual array values using a $ instead of a @.

You can assign several keys and values to a hash by placing their scalar values in a list separated by commas and surrounded by a pair of parentheses. Each successive pair of scalars becomes a key and a value in the hash. For instance, you can assign a hash the empty list:

```
%hash = (  );
```

You can also assign one or more scalar key/value pairs:

```
%genes_by_name = ('gene1', 'AACCCGGTTGGTT', 'gene2', 'CCTTTCGGAAGGTC');
```

There is an another way to do the same thing, which makes the key/value pairs more readily apparent. This accomplishes the same thing as the preceding example:

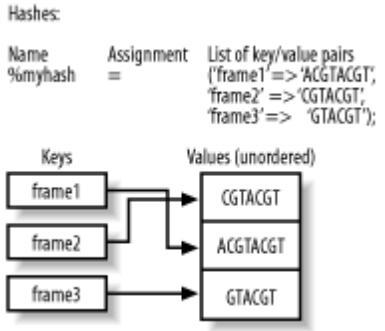
```
%genes_by_name = (
    'gene1' => 'AACCCGGTTGGTT',
    'gene2' => 'CCTTTCGGAAGGTC'
);
```

To get the value associated with a particular key, precede the hash name with a $ and follow it with a pair of curly braces containing the scalar value of the key:

```
$genes_by_name{'gene1'}
```

This returns the value 'AACCCGGTTGGTT', given the value previously assigned to the key 'gene1' in the hash %genes_by_name. Figure A-2 shows a hash with three keys.

Figure A-2. Schematic of a hash



You can get an array of all the keys in a hash with the operator "keys", and you can get an array of all the values in a hash with the operator "values".

A.8 Complex Data Structures

The standard Perl data types, scalar, array, and hash, can be combined into more complex data structures using references. The construction of complex data structures is summarized in Chapter 2.

As an example, you can define a two-dimensional matrix as an array of anonymous arrays (recall that an anonymous array is a reference to array data):

```
@microarray = (
    [ 10, 2, 14 ],
    [ 15, 4, 54 ],
    [ 51, 0, 99 ]
);
```


A.9 Operators

Operators are functions that represent basic operations on values: addition, subtraction, etc. They are frequently used and are core parts of the Perl programming language. They are really just functions that take arguments. For instance, + is the operator that adds two numbers, like so:

```
3 + 4;
```

Operators typically have one, two, or three operands; in the example just given, there are two operands 3 and 4.

Operators can appear before, between, or after their operands. For example, the plus operator + appears between its operands.

A.10 Operator Precedence

Operator precedence determines the order in which the operations are applied. For instance, in Perl, the expression:

```
3 + 3 * 4
```

isn't evaluated left to right, which calculates 3 + 3 equals 6, and 6 times 4 results in a value of 24; the precedence rules cause the multiplication to be applied first, for a final result of 15. The precedence rules are available in the perlop manpage and in most Perl books. However, I recommend you use parentheses to make your code more readable and to avoid bugs. They make the following expressions unambiguous; the first:

```
(3 + 3) * 4
```

evaluates to 24, and the second:

```
3 + (3 * 4)
```

evaluates to 15.

A.11 Basic Operators

For more information on how operators work, consult the perlop documentation bundled with Perl.

A.11.1 Arithmetic Operators

Perl has the basic five arithmetic operators:
+

Addition
-

Subtraction
*

Multiplication
/

Division
**

Exponentiation

These operators work on both integers and floating-point values (and if you're not careful, strings as well).

Perl also has a modulus operator, which computes the remainder of two integers:

```
% modulus
```

For example, 17 % 3 is 2, because 2 is left over when you divide 3 into 17.

Perl also has autoincrement and autodecrement operators:

```
++ add one
-- subtract one
```

Unlike the previous six operators, these change a variable's value. $x++ adds one to $x, changing 4 to 5 (or a to b).

A.11.2 Bitwise Operators

All scalars, whether numbers or strings, are represented as sequences of individual bits "under the hood." Every once in a while, you need to manipulate those bits, and Perl provides five operators to help:
&

Bitwise and
|

Bitwise or
^

Bitwise xor

A.12 Conditionals and Logical Operators

This section covers conditional statements and logical operators.

A.12.1 true and false

In a conditional test, an expression evaluates to true or false, and based on the result, a statement or block may or may not be executed.

A scalar value can be true or false in a conditional. A string is false if it's the empty string (represented as "" or ''). A string is true if it's not the empty string.

Similarly, an array or a hash is false if empty and true if nonempty.

A number is false if it's 0; a number is true if it's not 0.

Most things you evaluate in Perl return some value (such as a number from an arithmetic expression or an array returned from a subroutine), so you can use most things in Perl in conditional tests. Sometimes you may get an undefined value, for example, if you try to add a number to a variable that has not been assigned a value. Things might then fail to work as expected. For instance, the following:

```
use strict;
use warnings;
my $a;
my $b;
$b = $a + 2;
```

produces the warning output:

```
Use of uninitialized value in addition (+) at - line 5.
```

You can test for defined and undefined values with the Perl function defined.

A.12.2 Logical Operators

There are four logical operators:
notandorxor

not turns true values into false and false values into true. Its use is best illustrated in code:

```
if(not $done) {...}
```

This executes the code only if $done is false.

and is a binary operator that returns true if both its operands are true. If one or both of the operands are false, the operator returns false:

```
1   and  1    returns true
'a' and  ''   returns false
''  and  0    returns false
```

A.13 Binding Operators

Binding operators are used for pattern matching, substitution, and transliteration on strings. They are used with regular expressions that specify the patterns:

```
'ACGTACGTACGTACGT' =~ /CTA/
```

The pattern is the string CTA, enclosed by forward slashes. The string binding operator is =~; it tells the program which string to search, returning true if the pattern appears in the string.

!~ is another string binding operator; it returns true if the pattern isn't in the string:

```
'ACGTACGTACGTACGT' !~ /CTA/
```

This is equivalent to:

```
not 'ACGTACGTACGTACGT' =~ /CTA/
```

You can substitute one pattern for another using the string binding operator. In the next example, s/thine/nine/ is the substitution command, which substitutes the first occurrence of thine with the string nine:

```
$poor_richard = 'A stitch in time saves thine.';
$poor_richard =~ s/thine/nine/;
print $poor_richard;
```

This produces the output:

```
A stitch in time saves nine.
```

Finally, the transliteration (or translate) operator tr substitutes characters in a string. It has several uses, but the two uses I've covered are first, to change bases to their complements A ⟶ T, C ⟶ G, G ⟶ C, and T ⟶ A:

```
$DNA = 'ACGTTTAA';
$DNA =~ tr/ACGT/TGCA/;
```

This produces the value:

```
TGCAAATT
```

Second, the tr operator counts the number of a particular character in a string, as in this example which counts the number of Gs in a string of DNA sequence data:

```
$DNA = 'ACGTTTAA';
$count = ($DNA =~ tr/A//);
print $count;
```

This produces the value 3; it shows that a pattern match can return a count of the number of translations made in a string, which is then assigned to the variable $count.

A.14 Loops

Loops repeatedly execute the statements in a block until a conditional test changes value. There are several forms of loops in Perl:

```
while(CONDITION) {BLOCK}
until(CONDITION) {BLOCK}
for(INITIALIZATION ; CONDITION ; RE-INITIALIZATION ) {BLOCK}
foreach VAR (LIST) {BLOCK}
for VAR (LIST) {BLOCK}
do {BLOCK} while (CONDITION)
do {BLOCK} until (CONDITION)
```

The while loop first tests if the conditional is true; if so, it executes the block and then returns to the conditional to repeat the process. If false, it does nothing, and the loop is over:

```
$i = 3;
while ( $i ) {
    print "$i\n";
    $i--;
}
```

This produces the output:

```
3
2
1
```

Here's how the loop works. The scalar variable $i is first initialized to 3 (this isn't part of the loop). The loop is then entered, and $i is tested to see if it has a true (nonzero) value. It does, so the number 3 is printed, and the decrement operator is applied to $i, which reduces its value to 2. The block is now over, and the loop starts again with the conditional test. It succeeds with the true value 2, which is printed and decremented. The loop restarts with a test of $i, which is now the true value 1; 1 is printed and decremented to 0. The loop starts again; 0 is tested to see if it's true, and it's not, so the loop is now finished.

Loops often follow the same pattern, in which a variable is set, and a loop is called, which tests the variable's value and then executes a block, which includes changing the value of the variable.

The for loop makes this easy by including the variable initialization and the variable change in the loop statement. The following is exactly equivalent to the preceding example and produces the same output:

```
for ( $i = 3 ; $i ; $i-- ) {
    print "$i\n";
}
```

The foreach loop is a convenient way to iterate through the elements in an array. Here's an example:

```
@array = ('one', 'two', 'three');

foreach $element (@array) {
    print $element\n";
}
```

This prints the output:

```
one
two
three
```

The foreach loop specifies a scalar variable $element to be set to each element of the array. (You may use any variable name or none, in which case the special variable $_ is used automatically.) The array to be iterated over is

A.15 Input/Output

This section covers getting information into programs and receiving data back from them.

A.15.1 Input from Files

Perl has several convenient ways to get information into a program. The non-CGI programs in this book usually get input by opening and reading files. I've emphasized this way of getting input because it behaves very much the same way on any computer you may be using. You've observed the open and close system calls and how to associate a filehandle with a file when you open it, which then is used to read in the data. As an example:

```
open(FILEHANDLE, "informationfile");
@data_from_informationfile = <FILEHANDLE>;
close(FILEHANDLE);
```

This code opens the file informationfile and associates the filehandle FILEHANDLE with it. The filehandle is then used within angle brackets to actually read in the contents of the file and store the contents in the array @data_from_informationfile. Finally, the file is closed by referring once again to the opened filehandle.

A.15.2 Input from STDIN

Perl allows you to read in any input that is automatically sent to your program via standard input (STDIN). STDIN is a filehandle that by default is always open. Your program may be expecting some input that way. For instance, on a Mac, you can drag and drop a file icon onto the Perl applet for your program to make the file's contents appear in STDIN. On Unix systems, you can pipe the output of some other program into the STDIN of your program with shell commands such as:

```
someprog | my_perl_program
```

You can also pipe the contents of a file into your program with:

```
cat file | my_perl_program
```

or with:

```
my_perl_program < file.
```

Your program can then read in the data (from program or file) that comes as STDIN just as if it came from a file that you've opened:

```
@data_from_stdin = <STDIN>;
```

A.15.3 Input from Files Named on the Command Line

You can name your input files on the command line. <> is shorthand for <ARGV>. The ARGV filehandle treats the array @ARGV as a list of filenames and returns the contents of all those files, one line at a time. Perl places all command-line arguments into the array @ARGV. Some of these may be special flags, which should be read and removed from @ARGV if there will also be data files named. Perl assumes that anything in @ARGV refers to an input filename when it reaches a < > command. The contents of the file or files are then available to the program using the angle brackets without a filehandle, like so:

```
@data_from_files = <>;
```

For example, on Microsoft, Unix, or on the Mac OS X, you specify input files at the command line, like so:

```
% my_program file1 file2 file3
```

A.15.4 Output Commands

A.16 Regular Expressions

Regular expressions are, in effect, an extra language that lives inside the Perl language. In Perl, they have quite a lot of features. First, I'll summarize how regular expressions work in Perl; then, I'll present some of their many features.

A.16.1 Overview

Regular expressions describe patterns in strings. The pattern described by a single regular expression may match many different strings.

Regular expressions are used in pattern matching, that is, when you look to see if a certain pattern exists in a string. They can also change strings, as with the s/// operator that substitutes the pattern, if found, for a replacement. Additionally, they are used in the tr function that can transliterate several characters into replacement characters throughout a string. Regular expressions are case-sensitive, unless explicitly told otherwise.

The simplest pattern match is a string that matches itself. For instance, to see if the pattern 'abc' appears in the string 'abcdefghijklmnopqrstuvwxyz', write the following in Perl:

```
$alphabet = 'abcdefghijklmnopqrstuvwxyz';
if( $alphabet =~ /abc/ ) {
    print $&;
}
```

The =~ operator binds a pattern match to a string. /abc/ is the pattern abc, enclosed in forward slashes to indicate that it's a regular-expression pattern. $& is set to the matched pattern, if any. In this case, the match succeeds, since 'abc' appears in the string $alphabet, and the code just given prints out abc.

Regular expressions are made from two kinds of characters. Many characters, such as 'a' or 'Z', match themselves. Metacharacters have a special meaning in the regular-expression language. For instance, parentheses are used to group other characters and don't match themselves. If you want to match a metacharacter such as (in a string, you have to precede it with the backslash metacharacter \(in the pattern.

There are three basic ideas behind regular expressions. The first is concatenation: two items next to each other in a regular-expression pattern (that's the string between the forward slashes in the examples) must match two items next to each other in the string being matched (the $alphabet in the examples). So, to match 'abc' followed by 'def', concatenate them in the regular expression:

```
$alphabet = 'abcdefghijklmnopqrstuvwxyz';
if( $alphabet =~ /abcdef/ ) {
        print $&;
}
```

This prints:

```
abcdef
```

The second major idea is alternation. Items separated by the | metacharacter match any one of the items. For example, the following:

```
$alphabet = 'abcdefghijklmnopqrstuvwxyz';
if( $alphabet =~ /a(b|c|d)c/ ) {
        print $&;
}
```

A.17 Scalar and List Context

Every operation in Perl is evaluated in either scalar or list context. Many operators behave differently depending on the context they are in, returning a list in list context and a scalar in scalar context.

The simplest example of scalar and list contexts is the assignment statement. If the left side (the variable being assigned a value) is a scalar variable, the right side (the values being assigned) are evaluated in scalar context. In the following examples, the right side is an array @array of two elements. When the left side is a scalar variable, it causes @array to be evaluated in scalar context. In scalar context, an array returns the number of elements in an array:

```
@array = ('one', 'two');
$a = @array;
print $a;
```

This prints:

```
2
```

If you put parentheses around the $a, you make it a list with one element, which causes @array to be evaluated in list context:

```
@array = ('one', 'two');
($a) = @array;
print $a;
```

This prints:

```
one
```

Notice that when assigning to a list, if there are not enough variables for all the values, the extra values are simply discarded. To capture all the variables, you'd do this:

```
@array = ('one', 'two');
($a, $b) = @array;
print "$a $b";
```

This prints:

```
one two
```

Similarly, if you have too many variables on the left for the number of right variables, the extra variables are assigned the undefined value undef.

When reading about Perl functions and operations, notice what the documentation has to say about scalar and list context. Very often, if your program is behaving strangely, it's because it is evaluating in a different context than you had thought.

Here are some general guidelines on when to expect scalar or list context:

- You get list context from function calls (anything in the argument position is evaluated in list context) and from list assignments.
- You get scalar context from string and number operators (arguments to such operators as . and + are assumed to be scalars); from boolean tests such as the conditional of an if () statement or the arguments to the || logical operator; and from scalar assignment.

A.18 Subroutines

Subroutines are defined by the keyword sub, followed by the name of the subroutine, followed by a block enclosed by curly braces containing the body of the subroutine. The following is a simple example.

```
sub a_subroutine {
    print "I'm in a subroutine\n";
}
```

In general, you can call subroutines using the name of the subroutine followed by a parenthesized list of arguments:

```
a_subroutine();
```

Arguments can be passed into subroutines as a list of scalars. If any arrays are given as arguments, their elements are interpolated into the list of scalars. The subroutine receives all scalar values as a list in the special variable @_. This example illustrates a subroutine definition and the calling of the subroutine with some arguments:

```
sub concatenate_dna {
    my($dna1, $dna2) = @_;

    my($concatenation);

    $concatenation = "$dna1$dna2";

    return $concatenation;
}

print concatenate_dna('AAA', 'CGC');
```

This prints:

```
AAACGC
```

The arguments 'AAA' and 'CGC' are passed into the subroutine as a list of scalars. The first statement in the subroutine's block:

```
  my($dna1, $dna2) = @_;
```

assigns this list, available in the special variable @_, to the variables $dna1 and $dna2.

The variables $dna1 and $dna2 are declared as my variables to keep them local to the subroutine's block. In general, you declare all variables as my variables; this can be enforced by adding the statement use strict; near the beginning of your program. However, it is possible to use global variables that are not declared with my, which can be used anywhere in a program, including within subroutines.

The statement:

```
  my($concatenation);
```

declares another variable for use by the subroutine.

After the statement:

```
$concatenation = "$dna1$dna2";
```

performs the work of the subroutine, the subroutine defines its value with the return statement:

```
  return $concatenation;
```

The value returned from a call to a subroutine can be used however you wish; in this example, it is given as the argument to the print function.

A.19 Modules and Packages

Chapter 1 summarizes the basic concepts of namespaces, packages, and modules.

A namespace is a table containing the names and values of variables and subroutines. Namespaces are excellent ways to protect one part of your code from unintentionally using the same variable or subroutine name as appears in another part of your code, causing a namespace collision and often leading to incorrect (but hard to identify) program behavior.

By default, a Perl program uses the namespace called main.

The package declaration enables you to declare and use different namespaces for different parts of your program. For instance, to declare a new namespace called Outer, use the following statement:

```
package Outer;
```

Package declarations usually occur at or near the top of a file and are in effect throughout the file, but they can appear several times within a file, causing the active namespace to switch each time they are called.

When a file has one package declaration at the top of the file and it's named with the package name followed by the .pm suffix (e.g, Outer.pm), the file is called a Perl module. (The module also needs to end with the statement "1;" to load correctly when called.)

The code for a Perl module can be used in a Perl program by referencing the file defining the module with a use statement, as in the following example:

```
use Outer;
```

The Perl interpreter will then try to find a file called Outer.pm.

Chapter 1 gives the basic details on how to manage modules so that the Perl interpreter can find them.

A.20 Object-Oriented Programming

Object-oriented Perl programming is introduced in Chapter 3 and Chapter 4, and used in all the remaining chapters of the book.

There are three main concepts:

- A class is a Perl module with a package declaration. The name of the class is the same as the name of the package; the module file also has the same name but with ".pm" (for Perl module) appended.
- An object is a reference to a collection of data used by a Perl class, usually but not necessarily, implemented as a hash. An object is marked with the name of its class using bless.
- A method is a subroutine in the class.

Arrow notation -> is used in a special way in object-oriented Perl code and has an effect on what arguments are passed to the class methods.

You use a class in your code by saying, for example:

```
use Goodclass;
```

You create an object by calling a constructor method in the class, which is usually (but not necessarily) called new, for example:

```
$goodobject = Goodclass->new( parameter1 => 1, parameter2 => 2);
```

Note that arguments are usually (but not necessarily) specified using the hash notation key => value, and therefore can be given in any order.

You call other methods in the class by invoking them from a class object. For example you call the method stuff in class Goodclass on a Goodclass object $goodobject, with argument type initialized to the value 'good', and save its output in the array @goodstuff, like so:

```
@goodstuff = $goodobject->stuff(type => 'good');
```

The arrow notation causes the called method to insert an additional argument. (In the examples just shown, the methods are the subroutines new and stuff).

The additional argument is the class information that appears to the left of the arrow. So, in these examples, the method new has the argument list:

```
('Goodclass', parameter1=1, parameter2=2)
```

The method stuff has the argument list:

```
($goodobject, type='good')
```

The details of how arguments are passed to methods are important to know only if you are writing a class, not if you are using one. If you simply use a class, you only have to know how to call the class methods using the arrow

A.21 Built-in Functions

Perl has many built-in functions. Table A-6 is a partial list with short descriptions.

Table A-6. Perl built-in functions

Function	Summary
abs VALUE	Return the absolute value of its numeric argument
atan2 Y, X	Return the principal value of the arc tangent of Y/X from -? to ?
chdir EXPR	Change the working directory to EXPR (or home directory by default)
chmod MODE LIST	Change the file permissions of the LIST of files to MODE
chomp (VARIABLE or LIST)	Remove ending newline from string(s), if present
chop (VARIABLE or LIST)	Remove ending character from string(s)
chown UID, GID, LIST	Change owner and group of LIST of files to numeric UID and GID
close FILEHANDLE	Close the file, socket, or pipe associated with FILEHANDLE
closedir DIRHANDLE	Close the directory associated with DIRHANDLE
cos EXPR	Return the cosine of the radian number EXPR
dbmclose HASH	Break the binding between a DBM file and a hash

Appendix B. Installing Perl

Perl is a popular programming language that is extensively used in areas such as bioinformatics and web programming. It's become popular with biologists because it is so well suited to several bioinformatics tasks. This appendix is geared to those who are attempting the language for the first time.

Perl is also an application and is available (at no cost) to run on all operating systems found in the average biology lab (Unix and Linux, Macintosh, Windows, VMS, and more). The Perl application on your computer takes a Perl language program (such as one of the programs in this book), translates it into instructions the computer can understand, and runs (or executes) it.

The word Perl, then, refers both to the language in which you write programs and to the application on your computer that runs those programs. You can always tell from context which of these two meanings is being used.

Every computer language such as Perl needs to have a translator application (an interpreter or compiler) that can turn programs into instructions the computer can actually run. The Perl application is often referred to as the Perl interpreter, and it includes a Perl compiler as well. You will also see Perl programs referred to as Perl scripts or Perl code.

B.1 Installing Perl on Your Computer

Here are the basic steps for installing Perl on your computer:

1. Check to see if Perl is already installed; if so, check the version.
2. Get Internet access and go to the Perl home page at http://www.perl.com.
3. Go to the Downloads page and determine which Perl distribution to download.
4. Download the Perl distribution.
5. Install the distribution on your computer.

B.1.1 Perl May Already Be Installed

Many computers—especially Unix and Linux computers—come with Perl already installed. (Note that Unix and Linux are essentially the same thing, as far as the operating system is concerned; Linux is a clone, or functional copy, of a Unix system.) So, first check to see if Perl is already there. On Unix and Linux, type the following at a command prompt:

```
$ perl -v
```

If Perl is already installed, you'll see a message something like this:

```
This is perl, v5.8.0 built for i686-linux

Copyright 1987-2002, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using 'man perl' or 'perldoc perl'.  If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

If Perl isn't installed, you'll get a message something like this:

```
perl: command not found
```

If you're on a shared Unix system, at a university or business, check with the system administrator if this fails, because although Perl may be installed, your environment may not be set to find it. (Or, the system administrator may say, "You need Perl? Okay, I'll install it for you!")

On Windows or Macintosh, look at the program menus, or use the find program to search for perl. You can also try typing perl -v at an MS-DOS command window or at a shell window on the Mac OS X. (Note that the Mac OS X is a Unix system!)

B.2 Versions of Perl

Perl, like almost all popular software, has gone through much growth and change over the course of its 15-year life. The authors, Larry Wall and a large group of cohorts, publish new versions periodically. These new versions have been carefully designed to support most programs written under old versions, but occasionally some major new features are added that simply won't work with older versions of Perl.

This book assumes you have Perl Version 5 or higher installed. It's likely that if you have Perl already installed on your computer, it's Perl 5. But it's best to check. On a Unix or Linux system, or from an MS-DOS or Mac OS X command window, the *perl -v* command just shown displays the version number, in my case Version 5.8.0. The number 5.8.0 is "bigger" than 5, so I'm okay. If you get a smaller number (very likely 4.036), you'll have to install a recent version of Perl to enable the majority of programs in this book to run as shown.

What about future versions? Perl is always evolving, and Perl Version 6 is on the horizon. Will the code in this book still work in Perl 6? The answer is yes. Although Perl 6 is going to add some new things to the language, it should have no trouble with the Perl 5 code in this book.

B.3 Internet Access

To install Perl, you will need to download it from the Internet, as shown in the next few pages.

If you don't have Internet access, you can try taking your computer to a friend who does have such access, and connecting long enough to install Perl. You can also use a Zip drive or burn a CD from a friend's computer to bring the Perl software to your computer. There are also commercial shrink-wrapped CDs of Perl available from several sources (ask at your local software store), and several books include CDs with Perl.

B.4 Downloading

Following are some directions for getting Perl to work on your Unix or Linux, Macintosh, and Windows computers. (It is also available for several other less common operating systems from the Perl web site.)

There is one web site that serves as a central jumping off point for all things Perl: http://www.perl.com. This web page has a Downloads clickable button that guides you to everything you need to install Perl on your computer.

The more specific directions that follow are up-to-date as of this writing, but you should be aware that the web pages they refer to may change their design. So, basically, go to http://www.perl.com, and look for the Downloads link to click on. The system should tell you everything you need to know after that. There will be a HELP link and other helpful links besides. So even if the information in this section becomes outdated, you will certainly be able to visit the main Perl web site and find all you need to install Perl.

Downloading and installing Perl is usually quite easy; in fact, the majority of the time it's perfectly painless, but sometimes you may have to put some effort into getting it to work. If you're new at programming, and you run into difficulties, the best thing to do is to ask for help from someone who is a professional computer programmer and/or administrator, teacher, or someone in the lab who already programs in Perl. But the chances are that you won't need any help: keep reading!

B.4.1 Binary Versus Source Code

One choice you will find when downloading is the choice between binary or source code distributions of Perl. The chances are very good that a binary version will be available for your computer. Get that if it's available for your particular system.

Recall that a binary (or executable or compiled program) is a program that has been translated into machine language and is ready to run. Source code is a program written in a language such as C or Perl, which can then be compiled or interpreted to become a binary. The Perl application is actually written in the C programming language, so it's also possible to get the C source code for the Perl application and compile it to create the Perl application binary.

The best choice for installing Perl on your computer is usually to get an already made binary version of the program, because nothing else needs to be done. However, if no binary is available, or if you want to control the various options of your Perl installation, you can get the source code for Perl, which is itself written in the C programming language. You then compile it using a C compiler. But I repeat: see if you can find a binary for your particular computer operating system; compiling from source code is more complicated for beginners! Details are available at the Perl web site.

B.4.2 Perl for Unix and Linux

Recall that Unix and Linux are essentially the same kind of operating system—Linux is a clone of Unix. Both Unix and Linux come in several variants offered by various companies.

Perl was originally developed on Unix and for quite some time now it has come already installed on most such systems. Open a window and type *perl -v*. If you get version information, Perl is there.

B.5 How to Run Perl Programs

The details of how to run Perl vary depending on the operating system of your computer. The instructions that come with the version of Perl you install on your computer contain all you need to know. I'll just give a short summary here that will point you in the right direction.

There is a part of the Perl documentation called perlrun. You can find it at http://www.perl.com/pub/doc/manual/html/pod/perlrun.html. It gives all the options of running Perl, especially on Unix and Linux; it's complete but not for the beginner.

B.5.1 Running Perl Programs on Unix or Linux

On Unix or Linux, you usually run Perl programs from the command line. You can run a Perl program in a file called this_program by typing:

```
perl this_program
```

if you're in the same directory as the program. If you're not in the same directory, you may have to give the pathname of the program:

```
perl /usr/local/bin/this_program
```

Usually, you set the first line of *this_program* to have the correct pathname for Perl on your system, since different machines may have installed Perl in different directories. On my computer, I use the following as the first line of my Perl programs:

```
#!/usr/bin/perl
```

You can type *which perl* to find the pathname where Perl is installed on your system.

You also usually make the program executable, using the chmod program:

```
chmod 755 this_program
```

If you've set the first line correctly and used *chmod*, you can just type the name of the Perl program to run it. So, if you're in the same directory as the program, you can type *./this_program* or, if the program is in a directory that's included in your $PATH or $path variable, you can type *this_program*.[1]

[1] $PATH is the variable used for the shells sh, bash, and ksh; $path is the variable used for *csh*, *tcsh*, and so on.

If your Perl program won't run, the error messages you get from the shell in the command window may be a little confusing. For instance, the bash shell on my Linux system gives the error message:

```
bash: ./my_program: No such file or directory
```

in two cases: if there really is no program called my_program in the current directory, or if the first line of my_program has incorrectly given the location of Perl. So watch for that, especially when running programs from CPAN that may have different pathnames for Perl embedded in their first lines. Also, if you type *my_program*, you may get the error message:

```
bash: my_program: command not found
```

which means that the operating system can't find the program. But it's right there in your directory! The problem is probably that your $PATH or $path variable doesn't include the current directory, so that the system isn't even looking in the current directory for the program. In this case, change the $PATH or $path variable (depending on which shell you're using); or just type *./my_program* instead of my_program.

B.6 Finding Help

Make sure you have any available documentation. If you install Perl as outlined earlier, this is already done as part of the general Perl installation, and the instructions that come with your Perl distribution will explain how to get at the documentation on your system. There is also excellent online Internet documentation, which can be found at http://www.perl.com.

First, point your web browser at the main Perl web site, bookmark it, and look around a little, especially at the available documentation. This is an essential resource. Also, point your browser at http://www.ncbi.nlm.nih.gov (the National Center for Biotechnology Information) and http://www.ebi.ac.uk/ (the European Bioinformatics Institute) for two of the biggest government-sponsored bioinformatics resources. These are among the most important web sites for Perl and bioinformatics.

Also very useful is the standard book Programming Perl (now in its third edition). You can do fine with the (free) online documentation, but if you end up doing a lot of Perl programming, Programming Perl (and perhaps a few others) will probably end up on your bookshelf.

Most languages have a standard document set that includes the whole story about the language definition and use. In Perl, this is included with the program as the on-line manual. Although programming manuals often suffer from poor writing, it is best to be prepared to dig into them. A well-honed ability to skim is a great asset. The Perl manual isn't bad; its main problem, that it shares with most manuals, is that all the details are in there, so it can be a bit overwhelming at first. However, the Perl documentation does a decent job of helping the beginner navigate, by means of tutorial documents.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of Mastering Perl for Bioinformatics is a North American bullfrog (Rana catesbeiana). It is native to the central and eastern United States, as well as southern portions of Canada. However, this bullfrog has since been introduced as far away from its native habitat as Asia, Europe, and Hawaii.

The North American bullfrog is the largest true frog in North America and can weigh over a pound. It can grow up to eight inches in length, although the norm is four to five inches. The gender of the bullfrog is ascertained by comparing the size of the external ear (the tympanum) relative to the size of the eye.

Bullfrogs are predators and also are cannibalistic. Their role in the environment is to control the population of insect pests as well as snakes and mice. In fact, the zeal of the North American bullfrog threatens to drive other frog species to extinction.

The generic name (rana) comes from the Latin for frog, while the species name (catesbeiana) honors an English naturalist. In the 18th century, Mark Catesby (1683-1749) produced the authoritative and exhaustive record of the flora and fauna found in the New World. Wealthy patrons in England eagerly received Catesby's regular shipments of specimens, including plants, birds, reptiles, insects, and frogs.

Mary Anne Weeks Mayo was the production editor and copyeditor, and Marlowe Shaeffer was the proofreader, for Mastering Perl for Bioinformatics. Jane Ellin and Colleen Gorman provided quality control. Marlowe Shaeffer, Mary Agner, and James Quill provided production assistance. John Bickelhaupt wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Andrew Savikas to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Reg Aubry.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.