Perl Programming for Bioinformatics

Mastering Perl for Bioinformatics

O'REILLY®

James D. Tisdall

Mastering Perl for Bioinformatics

Mastering Perl for Bioinformatics

James D. Tisdall

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

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.

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.[*] 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.

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).

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.

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

- 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
```

I couldn't find a way to list the options within my CPAN session, so I took a look at the CPAN documentation in another window with `perldoc CPAN`, searched for the string `prerequisites_policy`, and read the following:

```
prerequisites_policy
                   what to do if you are missing module prerequisites
                   ('follow' automatically, 'ask' me, or 'ignore')
```

To handle the prerequisites automatically, I configured my CPAN session like so:

```
cpan> o conf prerequisites_policy follow
```

Now, if you have a slow Internet connection, setting the option this way can be a problem, because some prerequisites may take a long time to install, and you'll be stuck waiting for everything to finish. In that case, you may prefer to be asked about each prerequisite, so you can see if you have the needed time to do the installation. Plan on the process taking at least a couple of hours. My experience using both a dial-up modem and a cable modem from a home office is that the time involved was not too onerous.

As you probably know by now, you can see a summary of CPAN session commands by typing `help`:

```
cpan> help

Display Information
 command  argument          description
 a,b,d,m  WORD or /REGEXP/  about authors, bundles, distributions, modules
 i        WORD or /REGEXP/  about anything of above
 r        NONE              reinstall recommendations
 ls       AUTHOR            about files in the author's directory

Download, Test, Make, Install...
 get                        download
 make                       make (implies get)
 test      MODULES,         make test (implies make)
 install   DISTS, BUNDLES   make install (implies test)
 clean                      make clean
 look                       open subshell in these dists' directories
 readme                     display these dists' README files

Other
 h,?           display this menu       ! perl-code   eval a perl command
 o conf [opt]  set and query options   q             quit the cpan shell
 reload cpan   load CPAN.pm again      reload index  load newer indices
 autobundle    Snapshot                force cmd     unconditionally do cmd
cpan>
```

To find the Bioperl distribution, I typed:

```
cpan> i /bioperl/
```

and got the following output:

```
cpan> i /bioperl/
CPAN: Storable loaded ok
Going to read /root/.cpan/Metadata
  Database was generated on Sun, 27 Apr 2003 04:12:50 GMT
  Bundle          Bundle::BioPerl (C/CR/CRAFFI/Bundle-BioPerl-2.04.tar.gz)
  Distribution    B/BI/BIRNEY/bioperl-0.05.1.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-0.6.2.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-0.7.0.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-1.0.2.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-1.0.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-1.2.1.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-1.2.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-db-0.1.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-ext-0.6.tar.gz
  Distribution    B/BI/BIRNEY/bioperl-gui-0.7.tar.gz
  Distribution    C/CR/CRAFFI/Bundle-BioPerl-2.04.tar.gz
  Module          Bio::LiveSeq::IO::BioPerl (B/BI/BIRNEY/bioperl-1.2.1.tar.gz)
  13 items found

cpan>
```

From looking at the Bioperl web page, I knew that the `Bundle::BioPerl` had extra useful modules in it that Bioperl uses.

I started the installation by first installing the `Bundle`:

```
cpan> install Bundle::BioPerl
```

This process fetched the module code from a CPAN repository, unpacked it, tested it, and installed it, including its prerequisites (without questions asked because I set the `follow` option as described earlier in this section).

Somewhat fortified by my success, I decided to go for broke and install the main Bioperl distribution.

From the search output from my CPAN query, `i /bioperl/`, just shown, I saw that the highest numbered release was 1.2.1. Just to be sure it was the latest release, I also spent some time at the Bioperl web site reading the news about the latest releases, so I was quite sure it was what I wanted.

I also spent some time reading the `INSTALL` file, and I knew that I was generally in good shape. I had a new-enough version of Perl (Version 5.8.0) and was on one of the standard supported platforms.

I installed it on a notebook computer with an Intel 686 processor that had the RedHat Linux 7.2 operating system. The computer and the operating system were about two years old, old enough that I did do some looking around on the Bioperl web site to see if there were any advisories about which versions of Linux were recommended or advised against.

In general, modern computer systems are complex, and they change rapidly. Operating systems and hardware have a replacement cycle of about two years. Some of this is planned obsolescence; some is legitimate technical progress. Whatever the cause, the result is that a system such as the one I'm describing needs several pieces to be in sync. The hardware, operating system software, Perl version, and Bioperl version have to coexist; other pieces such as the C compiler, web browser, web server, and so forth may also cause problems. Hence my interest in checking to see if there were any advisories on these topics.

However, I didn't see any warnings about my particular platform. And since I had recently installed the latest version of Perl with success, I felt I had performed due diligence and that I should go ahead and try to install the Bioperl modules.

To do so, I typed:

```
cpan> install B/BI/BIRNEY/bioperl-1.2.1.tar.gz
```

There followed a great amount of activity as CPAN got the distribution from the Internet and tested the various modules. After a time, the tests were complete; a few of them had failed, and CPAN decided not to install the modules. Since only a handful of module tests failed, I looked at the output on my screen and decided that the failures were only in peripheral parts of Bioperl I didn't have an immediate use for, and if I ever did need them, I could fix them later.

So, to finish the installation, I forced CPAN to do the installation despite the presence of some test failures:

```
cpan> force install B/BI/BIRNEY/bioperl-1.2.1.tar.gz
```

This resulted in the modules and the documentation being installed.

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.
```

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.

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;

# this script will only work with an internet connection
# on the computer it is run on

$seq_object = get_sequence('swissprot',"ROA1_HUMAN");

# uses the default database - nr in this case
$blast_result = blast_sequence($seq);

write_blast(">roa1.blast",$blast_report);
[tisdall]$ perl tut2.pl

-------------------- WARNING ---------------------
MSG: req was POST http://www.ncbi.nlm.nih.gov/blast/Blast.cgi
User-Agent: libwww-perl/5.69
Content-Length: 178
Content-Type: application/x-www-form-urlencoded

...
---------------------------------------------------
Submitted Blast for [blast-sequence-temp-id]
[tisdall]$ cat roa1.blast
[tisdall]$ ls -l roa1.blast
-rw-rw-r--    1 tisdall  tisdall         0 Apr 30 11:28 roa1.blast
[tisdall]$
```

Here, I experienced a problem. Running the program created a page of error output, mostly formatted in HTML, which I've truncated in the output. When I tried to cat the output file, there was nothing in it, which I verified (on Linux) by examining the file with `ls -l`, which showed that it had 0 bytes in it.

This wasn't very encouraging; the second of the two short example scripts was failing.

Looking at the two programs `tut1.pl` and `tut2.pl`, you can see that the second is an extension of the first; after getting the sequence, the second program also tries to blast it, and this is where it is failing. It's running (printing those ... dots took a while because the program was waiting for a reply from NCBI over the Internet), but it's not printing out the BLAST report to the file `roa1.blast`. What's going wrong?

I started my investigation by looking at the documentation for the `Bio::Perl` module by typing:

```
perldoc Bio::Perl
```

and searching for the function that is failing, `blast_sequence`. Here's what I found:

```
blast_sequence

 Title   : blast_sequence
 Usage   : $blast_result = blast_sequence($seq)
           $blast_result = blast_sequence('MFVEGGTFASEDDDSASAEDE');

 Function: If the computer has Internet accessibility, blasts
           the sequence using the NCBI BLAST server against nrdb.

           It choose the flavour of BLAST on the basis of the sequence.

           This function uses Bio::Tools::Run::RemoteBlast, which itself
           use Bio::SearchIO - as soon as you want to more, check out
           these modules
 Returns : Bio::Search::Result::GenericResult.pm

 Args    : Either a string of protein letters or nucleotides, or a
           Bio::Seq object
```

That last sentence about the `Args` gave me a clue. I went back and looked at the failing program, which I placed in a file called `tut2.pl`, and, sure enough, the argument for the `blast_sequence` function, `$seq`, is neither a string of sequence nor a `Bio::Seq` object. In fact, it's not even defined! Clearly, what the tutorial writer meant instead of `$seq`, was `$seq_object`, which is defined and populated by the earlier `get_sequence` method call.

I was visited by a brainwave, and I decided to put a `use strict` and a `use warnings` into the program and to declare variables with `my` to see what ensued:

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

```
# this script will only work with an internet connection
# on the computer it is run on

my $seq_object = get_sequence('swissprot',"ROA1_HUMAN");

# uses the default database - nr in this case
my $blast_result = blast_sequence($seq);

write_blast(">roa1.blast",$blast_report);
[tisdall]$ perl tut2.pl
Global symbol "$seq" requires explicit package name at tut2.pl line 11.
Global symbol "$blast_report" requires explicit package name at tut2.pl line 13.
Execution of tut2.pl aborted due to compilation errors.
[tisdall]$
```

Well, that's pretty clear. The variables `$seq` and `$blast_report` are wrong; apparently, the author intended to reuse the variables `$seq_object` and `$blast_result` instead. So, I edited the file and ran it again:

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

# this script will only work with an internet connection
# on the computer it is run on

my $seq_object = get_sequence('swissprot',"ROA1_HUMAN");

# uses the default database - nr in this case
my $blast_result = blast_sequence($seq_object);

write_blast(">roa1.blast",$blast_result);
[tisdall]$ perl tut2.pl
[tisdall]$ perl tut2.pl
Submitted Blast for [ROA1_HUMAN] ..................
[tisdall]$ ls -l roa1.blast
-rw-rw-r--    1 tisdall  tisdall     56888 May  5 15:05 roa1.blast
[tisdall]$
```

Examining the `roa1.blast` file convinced me that the program had successfully called `blast`. I decided that was a success, albeit a qualified one: this was a spot in the documentation that could use a little attention, clearly. By the time you're reading this, it may well have been fixed.

Fourth Test

Now, let me show you one more test of my new Bioperl installation.

Looking at the `Bio::Perl` documentation, I found the following example and discussion at the very beginning.

```
Bio::Perl(3)   User Contributed Perl Documentation   Bio::Perl(3)

NAME
       Bio::Perl - Functional access to BioPerl for people who
       don't know objects

SYNOPSIS
         use Bio::Perl;

         # will guess file format from extension
         $seq_object = read_sequence($filename);

         # forces genbank format
         $seq_object = read_sequence($filename,'genbank');

         # reads an array of sequences
         @seq_object_array = read_all_sequences($filename,'fasta');

         # sequences are Bio::Seq objects, so the following methods work
         # for more info see L<Bio::Seq>, or do 'perldoc Bio/Seq.pm'

         print "Sequence name is ",$seq_object->display_id,"\n";
         print "Sequence acc  is ",$seq_object->accession_number,"\n";
         print "First 5 bases is ",$seq_object->subseq(1,5),"\n";

         # get the whole sequence as a single string

         $sequence_as_a_string = $seq_object->seq( );

         # writing sequences

         write_sequence(">$filename",'genbank',$seq_object);

         write_sequence(">$filename",'genbank',@seq_object_array);

         # making a new sequence from just strings you have
         # from something else

         $seq_object = new_sequence("ATTGGTTTGGGGACCCAATTTGTGTGTTATATGTA",
             "myname","AL12232");

         # getting a sequence from a database (assumes internet connection)

         $seq_object = get_sequence('swissprot',"ROA1_HUMAN");

         $seq_object = get_sequence('embl',"AI129902");

         $seq_object = get_sequence('genbank',"AI129902");

         # BLAST a sequence (assummes an internet connection)

         $blast_report = blast_sequence($seq_object);
```

```
          write_blast(">blast.out",$blast_report);

DESCRIPTION
       Easy first time access to BioPerl via functions

FEEDBACK
       Mailing Lists

       User feedback is an integral part of the evolution of this
       and other Bioperl modules. Send your comments and sugges-
       tions preferably to one of the Bioperl mailing lists.
       Your participation is much appreciated.

         bioperl-l@bio.perl.org

       Reporting Bugs

       Report bugs to the Bioperl bug tracking system to help us
       keep track the bugs and their resolution. Bug reports can
       be submitted via email or the web:

         bioperl-bugs@bio.perl.org
         http://bugzilla.bioperl.org/

AUTHOR - Ewan Birney
       Email bioperl-l@bio.perl.org

       Describe contact details here

APPENDIX
       The rest of the documentation details each of the object
       methods.  Internal methods are usually preceded with a _
...
```

I took the example code from the SYNOPSIS section and put it in a file called bp1.pl. I noticed that the code was not meant to be a running program. The variable $filename refers to some sequence file in the first line of code without being initialized, and then the same variable name is used in several other lines for different purposes. This is not an uncommon situation in such SYNOPSIS sections; the point is to show how individual calls can be made to methods in the class, not to demonstrate a complete working program (although sometimes the code can be run exactly as is).

I had to make some changes to make this code a working, runnable program. I declared the strict and warnings pragmas and declared each variable with my. I created variables for the different sequence filenames I needed as both input and output, and I made sure that the input sequence files were on disk and of the correct variety (for example, I created the array.fasta file with three FASTA headers and sequences one after the other). In the end I had this code:

```
use Bio::Perl;
use strict;
use warnings;
```

```
my $gbfilename = 'AI129902.genbank';

# will guess file format from extension
my $seq_object0 = read_sequence($gbfilename);

# forces genbank format
my $seq_object1 = read_sequence($gbfilename,'genbank');

my $fastafilename = 'array.fasta';

# reads an array of sequences
my @seq_object_array = read_all_sequences($fastafilename,'fasta');

# sequences are Bio::Seq objects, so the following methods work
# for more info see L<Bio::Seq>, or do 'perldoc Bio/Seq.pm'

print "Sequence name is ",$seq_object1->display_id,"\n";
print "Sequence acc  is ",$seq_object1->accession_number,"\n";
print "First 5 bases is ",$seq_object1->subseq(1,5),"\n";

# get the whole sequence as a single string

my $sequence_as_a_string = $seq_object1->seq( );

# writing sequences

my $gbfilenameout = 'bpout.genbank';

write_sequence(">$gbfilenameout",'genbank',$seq_object1);

write_sequence(">$gbfilenameout",'genbank',@seq_object_array);

# making a new sequence from just strings you have
# from something else

my $seq_object2 = new_sequence("ATTGGTTTGGGGACCCAATTTGTGTGTTATATGTA",
    "myname","AL12232");

# getting a sequence from a database (assumes internet connection)

my $seq_object3 = get_sequence('swissprot',"ROA1_HUMAN");

my $seq_object4 = get_sequence('embl',"AI129902");

my $seq_object5 = get_sequence('genbank',"AI129902");

# BLAST a sequence (assummes an internet connection)

my $blast_report = blast_sequence($seq_object3);

write_blast(">blast.out",$blast_report);
```

As you see, I didn't check on the output of all the method calls, but the real purpose of this test of my newly installed Bioperl modules was just to see if all the modules and methods could be found and would run when called.

As mentioned previously, I also added `use strict;` and `use warnings;` and declared all the variables with `my`. Often, you don't see that usage in this kind of documentation; it's not required in Perl, and it may distract some readers of the documentation from the main point of showing how the methods are called. This is not an unusual omission to find in Perl class documentation, but of course, it's recommended and often very helpful, as you saw in the last section.

So, I ran my slightly edited version of the example code from the beginning of the `Bio::Perl` manpage, with the following results:

```
[tisdall]$ perl bp1.pl
Sequence name is AI129902
Sequence acc  is AI129902
First 5 bases is CTCCG
Submitted Blast for [ROA1_HUMAN] .........
[tisdall]$
```

I looked at each of the input and output files and verified the expected contents. The following output from a listing of the files will give you an idea of what to expect (although you may get different results by running the program with different input files or database lookups):

```
-rw-rw-r--    1 tisdall  tisdall      2391 May  5 10:37 AI129902.genbank
-rw-rw-r--    1 tisdall  tisdall      2485 May  5 13:00 array.fasta
-rw-rw-r--    1 tisdall  tisdall      1513 May  5 13:02 bp1.pl
-rw-rw-r--    1 tisdall  tisdall      3653 May  5 13:02 bpout.genbank
-rw-rw-r--    1 tisdall  tisdall     56888 May  5 13:04 blast.out
```

Notice that at the end of the program I call the `blast_sequence` method on the protein sequence object `$seq_object3`. I discovered that trying to run the `blast_sequence` method on a nucleotide sequence object (such as `$seq_object5`) failed. Although the documentation for the method said that the sequence type would be examined and the appropriate BLAST program called (for example, `blastp` for protein sequence and `blastx` for nucleotide sequence, against the nr nonredundant protein database), it always seemed to call `blastp` no matter what the input sequence, and therefore it failed unless called with protein sequence as I had stored in `$seq_object3`. Perhaps this bug, a disconnect between the code and the documentation, has been fixed by the time you read this.

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.

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.

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 program in my own directory gave me an error because the program evidently requires a FASTA-formatted file called `dna1.fa`:

```
[tisdall]$ perl -w bptutorial.pl 1

Beginning sequence_manipulations and SeqIO example...

------------- EXCEPTION  -------------
MSG: Could not open t/data/dna1.fa: No such file or directory
STACK Bio::Root::IO::_initialize_io /usr/local/lib/perl5/site_perl/5.8.0/Bio/Root/IO.
pm:264
STACK Bio::SeqIO::_initialize /usr/local/lib/perl5/site_perl/5.8.0/Bio/SeqIO.pm:437
STACK Bio::SeqIO::fasta::_initialize /usr/local/lib/perl5/site_perl/5.8.0/Bio/SeqIO/
fasta.pm:80
STACK Bio::SeqIO::new /usr/local/lib/perl5/site_perl/5.8.0/Bio/SeqIO.pm:355
STACK Bio::SeqIO::new /usr/local/lib/perl5/site_perl/5.8.0/Bio/SeqIO.pm:368
STACK main::__ANON__ bptutorial.pl:2758
STACK toplevel bptutorial.pl:3933

--------------------------------------
[tisdall]$
```

I decided it would be easier to run `bptutorial.pl` from the distribution directory. I entered that directory; on my Linux machine, I unpacked the Bioperl source code into */usr/local/src/bioperl-1.2.1*. I then tried to run the demo:

```
[tisdall]$ cd /usr/local/src/bioperl-1.2.1
[tisdall]$ perl -w bptutorial.pl 1

Beginning sequence_manipulations and SeqIO example...
First sequence in fasta format...
>Test1
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTC
TGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGG
TCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTAC
ACAACATCCATGAAACGCATTAGCACCACC
Seq object display id is Test1
```

```
Sequence is
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTAC
CTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAAT
TACAGAGTACACAACATCCATGAAACGCATTAGCACCACC
Sequence from 5 to 10 is TTTCAT
Acc num is unknown
Moltype is dna
Primary id is Test1
Truncated Seq object sequence is TTTCAT
Reverse complemented sequence 5 to 10  is ATGAAA
Translated sequence 6 to 15 is LQRAICLCVD

Beginning 3-frame and alternate codon translation example...
ctgagaaaataa translated using method defaults   : LRK*
ctgagaaaataa translated as a coding region (CDS): MRK

Translating in all six frames:
 frame: 0 forward: LRK*
 frame: 0 reverse-complement: LFSQ
 frame: 1 forward: *EN
 frame: 1 reverse-complement: YFL
 frame: 2 forward: EKI
 frame: 2 reverse-complement: IFS
Translating with all codon tables using method defaults:
1 : LRK*
2 : L*K*
3 : TRK*
4 : LRK*
5 : LSK*
6 : LRKQ
9 : LSN*
10 : LRK*
11 : LRK*
12 : SRK*
13 : LGK*
14 : LSNY
15 : LRK*
16 : LRK*
21 : LSN*
[tisdall]$
```

That seemed to run without error. Now I wanted to see the Perl code that had run and produced that output.

Exploring a bit, I found that the POD documentation part of `bptutorial.pl` is roughly the first half of the file, and that the second half of the file contains the Perl code for the demos.

The author attributions and the libraries that are loaded are at the beginning of the Perl code section, somewhere near the middle of the `bptutorial.pl` file:

```
#!/usr/bin/perl

# PROGRAM  : bptutorial.pl
# PURPOSE  : Demonstrate various uses of the bioperl package
# AUTHOR   : Peter Schattner schattner@alum.mit.edu
# CREATED  : Dec 15 2000
# REVISION : $Id: ch09,v 1.31 2003/09/11 13:54:52 mam Exp mam $

use strict;
use Bio::SimpleAlign;
use Bio::AlignIO;
use Bio::SeqIO;
use Bio::Seq;
```

That seems like a good bet for a list of the most important Bioperl modules. Actually, some other modules are loaded for individual demos; at various points, the following come into play:

```
use Bio::SearchIO;
use Bio::Root::IO;
use Bio::MapIO;
use Bio::TreeIO;
use Bio::Perl
```

Following those use Module directives comes the following:

```
# subroutine references

my ($access_remote_db, $index_local_db, $fetch_local_db,
    $sequence_manipulations, $seqstats_and_seqwords,
    $restriction_and_sigcleave, $other_seq_utilities, $run_remoteblast,
    $run_standaloneblast,  $blast_parser, $bplite_parsing, $hmmer_parsing,
    $run_clustalw_tcoffee, $run_psw_bl2seq, $simplealign,
    $gene_prediction_parsing, $sequence_annotation, $largeseqs,
    $run_tree, $run_map, $run_struct, $run_perl, $searchio_parsing,
    $liveseqs, $demo_variations, $demo_xml, $display_help, $bpinspect1 );

# global variable file names.  Edit these if you want to try
#out a tutorial script on a different file

 Bio::Root::IO->catfile("t","data","ecolitst.fa");

# used in $sequence_manipulations
my $dna_seq_file = Bio::Root::IO->catfile("t","data","dna1.fa");

# used in $other_seq_utilities and in $run_perl and $sequence_annotation
my $amino_seq_file = Bio::Root::IO->catfile("t","data","cysprot1.fa");

# used in $blast_parser
my $blast_report_file = Bio::Root::IO->catfile("t","data","blast.report");

# used in $bplite_parsing
my $bp_parse_file1 = Bio::Root::IO->catfile("t","data","blast.report");

# used in $bplite_parsing
my $bp_parse_file2 = Bio::Root::IO->catfile("t","data","psiblastreport.out");

# used in $bplite_parsing
my $bp_parse_file3 = Bio::Root::IO->catfile("t","data","bl2seq.out");
```

```
# used in $run_clustalw_tcoffee
my $unaligned_amino_file = Bio::Root::IO->catfile("t","data","cysprot1a.fa");

# used in $simplealign
my $aligned_amino_file = Bio::Root::IO->catfile("t","data","testaln.pfam");

# other global variables
my (@runlist, $n );
```

A look at the documentation (`perldoc Bio::Root::IO`) shows that the catfile method returns the pathname of a file, which is being saved in a scalar variable such as `$dna_seq_file`. This method is there for portability between operating systems, because operating systems each have their own syntax for specifying pathnames.

After that comes the code for the help screen, the output of which you've already seen; next comes the individual demo subroutines; and finally the code for the main subroutine, which you saw in the last section.

At the very end of the `bptutorial.pl` file, I found the part of the code that launches all the demos:

```
## "main" program follows
#no strict 'refs';

    @runlist = @ARGV;
    # display help if no option
    if (scalar(@runlist)==0) {&$display_help;};
    # argument = 0 means run tests 1 thru 22
    if ($runlist[0] == 0) {@runlist = (1..22); };
    foreach $n  (@runlist) {
        if ($n ==100) {my $object = $runlist[1]; &$bpinspect1($object); last;}
        if ($n ==1) {&$sequence_manipulations; next;}
        if ($n ==2) {&$seqstats_and_seqwords; next;}
        if ($n ==3) {&$restriction_and_sigcleave; next;}
        if ($n ==4) {&$other_seq_utilities; next;}
        if ($n ==5) {&$run_perl; next;}
        if ($n ==6) {&$searchio_parsing; next;}
        if ($n ==7) {&$bplite_parsing; next;}
        if ($n ==8) {&$hmmer_parsing; next;}
        if ($n ==9) {&$simplealign ; next;}
        if ($n ==10) {&$gene_prediction_parsing; next;}
        if ($n ==11) {&$access_remote_db; next;}
        if ($n ==12) {&$index_local_db; next;}
        if ($n ==13) {&$fetch_local_db; next;}
        if ($n ==14) {&$sequence_annotation; next;}
        if ($n ==15) {&$largeseqs; next;}
        if ($n ==16) {&$liveseqs; next;}
        if ($n ==17) {&$run_struct; next;}
        if ($n ==18) {&$demo_variations; next;}
        if ($n ==19) {&$demo_xml; next;}
        if ($n ==20) {&$run_tree; next;}
        if ($n ==21) {&$run_map; next;}
        if ($n ==22) {&$run_remoteblast; next;}
        if ($n ==23) {&$run_standaloneblast; next;}
        if ($n ==24) {&$run_clustalw_tcoffee; next;}
        if ($n ==25) {&$run_psw_bl2seq; next;}
         &$display_help;
    }

## End of "main"
```

So, I searched for sequence_manipulation and found the following code for this, the first of the bptutorial.pl demos:

```
#################################################
# sequence_manipulations  ():
#

$sequence_manipulations = sub {

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

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


    # 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;

    # III.4 Manipulating individual sequences

    # The following methods return strings

    print "Seq object display id is ",
    $seqobj->display_id(), "\n"; # the human read-able id of the sequence
    print "Sequence is ",
    $seqobj->seq()," \n";        # string of sequence
    print "Sequence from 5 to 10 is ",
    $seqobj->subseq(5,10)," \n"; # part of the sequence as a string
    print "Acc num is ",
    $seqobj->accession_number(), " \n"; # when there, the accession number
    print "Moltype is ",
    $seqobj->alphabet(), " \n";    # one of 'dna','rna','protein'
    print "Primary id is ", $seqobj->primary_seq->primary_id()," \n";
    # a unique id for this sequence irregardless
    #print "Primary id is ", $seqobj->primary_id(), " \n";
    # a unique id for this sequence irregardless
    # of its display_id or accession number

    # The following methods return an array of  Bio::SeqFeature objects
    $seqobj->top_SeqFeatures; # The 'top level' sequence features
    $seqobj->all_SeqFeatures; # All sequence features, including sub
    # seq features

    # The following methods returns new sequence objects,
    # but do not transfer features across
    # truncation from 5 to 10 as new object
    print "Truncated Seq object sequence is ",
    $seqobj->trunc(5,10)->seq( ), " \n";
    # reverse complements sequence
    print "Reverse complemented sequence 5 to 10  is ",
    $seqobj->trunc(5,10)->revcom->seq, "  \n";
    # translation of the sequence
    print "Translated sequence 6 to 15 is ",
    $seqobj->translate->subseq(6,15), " \n";

    my $c = shift;
    $c ||= 'ctgagaaaataa';

    print "\nBeginning 3-frame and alternate codon translation example... \n";

    my $seq = new Bio::PrimarySeq('-SEQ' => $c,
                                  '-ID' => 'no.One');
    print "$c translated using method defaults   : ",
    $seq->translate->seq, "\n";

    # Bio::Seq uses same sequence methods as PrimarySeq
    my $seq2 = new Bio::Seq('-SEQ' => $c, '-ID' => 'no.Two');
    print "$c translated as a coding region (CDS): ",
    $seq2->translate(undef, undef, undef, undef, 1)->seq, "\n";

    print "\nTranslating in all six frames:\n";
    my @frames = (0, 1, 2);
    foreach my $frame (@frames) {
        print  " frame: ", $frame, " forward: ",
        $seq->translate(undef, undef, $frame)->seq, "\n";
        print  " frame: ", $frame, " reverse-complement: ",
        $seq->revcom->translate(undef, undef, $frame)->seq, "\n";
    }

    print "Translating with all codon tables using method defaults:\n";
    my @codontables = qw( 1 2 3 4 5 6 9 10 11 12 13 14 15 16 21 );
    foreach my $ct (@codontables) {
        print $ct, " : ",
        $seq->translate(undef, undef, undef, $ct)->seq, "\n";
    }

    return 1;
} ;

#################################################
```

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. This prints the first FASTA record from that file, as shown in the demo output in the last section and repeated here:

```
First sequence in fasta format...
>Test1
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTC
TGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGG
TCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTAC
ACAACATCCATGAAACGCATTAGCACCACC
```

The next part of the sequence_manipulation demo code shows many of the methods for extracting information from a sequence object:

```
# III.4 Manipulating individual sequences

# The following methods return strings

print "Seq object display id is ",
$seqobj->display_id(), "\n"; # the human read-able id of the sequence
print "Sequence is ",
$seqobj->seq()," \n";        # string of sequence
print "Sequence from 5 to 10 is ",
$seqobj->subseq(5,10)," \n"; # part of the sequence as a string
print "Acc num is ",
$seqobj->accession_number(), " \n"; # when there, the accession number
print "Moltype is ",
$seqobj->alphabet(), " \n";    # one of 'dna','rna','protein'
print "Primary id is ", $seqobj->primary_seq->primary_id()," \n";
# a unique id for this sequence irregardless
#print "Primary id is ", $seqobj->primary_id(), " \n";
# a unique id for this sequence irregardless
# of its display_id or accession number

# The following methods return an array of  Bio::SeqFeature objects
$seqobj->top_SeqFeatures; # The 'top level' sequence features
$seqobj->all_SeqFeatures; # All sequence features, including sub
# seq features

# The following methods returns new sequence objects,
# but do not transfer features across
# truncation from 5 to 10 as new object
print "Truncated Seq object sequence is ",
$seqobj->trunc(5,10)->seq( ), " \n";
# reverse complements sequence
print "Reverse complemented sequence 5 to 10  is ",
$seqobj->trunc(5,10)->revcom->seq, "  \n";
# translation of the sequence
print "Translated sequence 6 to 15 is ",
$seqobj->translate->subseq(6,15), " \n";
```

This section of the tutorial produced the following output (slightly reformatted to fit the pages of this book):

```
    Seq object display id is Test1
    Sequence is AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGA
    TAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACC
    AATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAAACGCATTAGCACCACC
    Sequence from 5 to 10 is TTTCAT
    Acc num is unknown
    Moltype is dna
    Primary id is Test1
    Truncated Seq object sequence is TTTCAT
    Reverse complemented sequence 5 to 10  is ATGAAA
    Translated sequence 6 to 15 is LQRAICLCVD
```

This part of the demo code is self-explanatory. As you can see, the methods return:

- The FASTA ID (the part immediately following the > sign in the first line)
- The sequence alone as a string (reformatted with line breaks to fit this book)
- The accession number if it is known (FASTA format won't have this, but Genbank format will, for instance)
- The type of molecule (dna, rna, or protein)
- A unique ID compared to other IDs in the running program but drawn from the file itself if possible
- The sequence of a new truncated sequence object defined from the original sequence object; a reverse complement
- The peptide resulting from a translation of a specified part of a sequence

The final section of the `sequence_manipulation` demo demonstrates the ability to translate nucleotides into all six reading frames using alternate codon translation tables if desired:

```
    my $c = shift;
    $c ||= 'ctgagaaaataa';

    print "\nBeginning 3-frame and alternate codon translation example... \n";

    my $seq = new Bio::PrimarySeq('-SEQ' => $c, '-ID' => 'no.One');
    print "$c translated using method defaults   : ",
    $seq->translate->seq, "\n";

    # Bio::Seq uses same sequence methods as PrimarySeq
    my $seq2 = new Bio::Seq('-SEQ' => $c, '-ID' => 'no.Two');
```

```
print "$c translated as a coding region (CDS): ",
$seq2->translate(undef, undef, undef, undef, 1)->seq, "\n";

print "\nTranslating in all six frames:\n";
my @frames = (0, 1, 2);
foreach my $frame (@frames) {
    print  " frame: ", $frame, " forward: ",
    $seq->translate(undef, undef, $frame)->seq, "\n";
    print  " frame: ", $frame, " reverse-complement: ",
    $seq->revcom->translate(undef, undef, $frame)->seq, "\n";
}

print "Translating with all codon tables using method defaults:\n";
my @codontables = qw( 1 2 3 4 5 6 9 10 11 12 13 14 15 16 21 );
foreach my $ct (@codontables) {
    print $ct, " : ",
    $seq->translate(undef, undef, undef, $ct)->seq, "\n";
}
```

This produces the output:

```
Beginning 3-frame and alternate codon translation example...
ctgagaaaataa translated using method defaults   : LRK*
ctgagaaaataa translated as a coding region (CDS): MRK

Translating in all six frames:
 frame: 0 forward: LRK*
 frame: 0 reverse-complement: LFSQ
 frame: 1 forward: *EN
 frame: 1 reverse-complement: YFL
 frame: 2 forward: EKI
 frame: 2 reverse-complement: IFS
Translating with all codon tables using method defaults:
1 : LRK*
2 : L*K*
3 : TRK*
4 : LRK*
5 : LSK*
6 : LRKQ
9 : LSN*
10 : LRK*
11 : LRK*
12 : SRK*
13 : LGK*
14 : LSNY
15 : LRK*
16 : LRK*
21 : LSN*
```

Again, this code is fairly easy to follow, although you may find yourself turning to the documentation for some of the finer points when you try to use these objects and methods in your own code.

This part of the code starts by getting an argument of sequence data if one is available to be shifted off of the @_ argument array and into the variable $c; otherwise, it sets the variable $c to a preset sequence:

```
my $c = shift;
$c ||= 'ctgagaaaataa';
```

Briefly, the methods that follow in this part of the sequence_manipulation demo do the following:

- Call the PrimarySeq::new constructor to create a lightweight sequence object from $c that doesn't have the extra annotation often present in a standard sequence file (see perldoc Bio::PrimarySeq for the whole story)
- Translate the sequence (from $c)
- Call the Bio::Seq constructor to create the $seq2 sequence object (from the same sequence data in $seq)
- Translate the CDS in the sequence object $seq2
- Translate the sequence in all six reading frames
- Translate the sequence using all 21 defined codon tables

The translate method seems to have lots of interesting options. However, if you try to look up the documentation for this method, you may have difficulty finding it. The problem is that Bioperl classes often make considerable use of inheritance. Say that the code is calling a method on a Bio::PrimarySeq object, as do the following lines from the demo (somewhat separated in the original):

```
my $seq = new Bio::PrimarySeq('-SEQ' => $c, '-ID' => 'no.One');
$seq->translate(undef, undef, $frame)->seq, "\n";
```

You may try perldoc Bio::PrimarySeq and not find a discussion of the translate method because it is being inherited from some other class. And since multiple inheritance is possible, it can take considerable effort to track down this method documentation by figuring out the parent classes and searching the documentation.

There's a very convenient way to find the documentation for a method, even if it's only inherited—it's built into the bptutorial.pl script. In the example just mentioned, you ask for the list of methods used by the Bio::PrimarySeq method.

```
[tisdall]$ perl bptutorial.pl 100 Bio::PrimarySeq

 ***Methods for Object Bio::PrimarySeq ********

 Methods taken from package Bio::DescribableI
 description   display_name

 Methods taken from package Bio::IdentifiableI
 authority   lsid_string   namespace   namespace_string   object_id   version

 Methods taken from package Bio::PrimarySeq
 accession   direct_seq_set   validate_seq
 Methods taken from package Bio::PrimarySeqI
 accession_number   alphabet   can_call_new   desc   display_id   id
 is_circular   length   moltype   primary_id   revcom   seq
 subseq   translate   trunc

 Methods taken from package Bio::Root::Root
 DESTROY   debug   verbose

 Methods taken from package Bio::Root::RootI
 carp   confess   deprecated   new   stack_trace   stack_trace_dump
 throw   throw_not_implemented   warn   warn_not_implemented
[tisdall]$
```

Sure enough, there under the heading "Methods taken from package Bio::Primary-SeqI" appears the method `translate`. You should then look at the `Bio::PrimarySeqI` documentation and find the following method description:

```
         translate

Title   : translate
Usage   : $protein_seq_obj = $dna_seq_obj->translate
          #if full CDS expected:
          $protein_seq_obj = $cds_seq_obj->translate(undef,undef,undef,undef,1);
Function:

          Provides the translation of the DNA sequence using full
          IUPAC ambiguities in DNA/RNA and amino acid codes.

          The full CDS translation is identical to EMBL/TREMBL
          database translation. Note that the trailing terminator
          character is removed before returning the translation
          object.

          Note: if you set $dna_seq_obj->verbose(1) you will get a
          warning if the first codon is not a valid initiator.

Returns : A Bio::PrimarySeqI implementing object
Args    : character for terminator (optional) defaults to '*'
          character for unknown amino acid (optional) defaults to 'X'
          frame (optional) valid values 0, 1, 2, defaults to 0
          codon table id (optional) defaults to 1
          complete coding sequence expected, defaults to 0 (false)
          boolean, throw exception if not complete CDS (true) or
          defaults to warning (false)
```

You will want to explore at least a few more of the tutorials embedded in `bptutorial.pl`, as we've done here with the first of the tutorials.

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