UNIX - Command-Line Survival Guide

Files, directories, commands, text editors

Lincoln Stein

Lecture Notes

Workshop Problem Set

Problem #1

a. Log into your machine. What is the full path to your home directory?
b. What is the path to the home directory of user *lstein*?
c. What is the path to the home directory of *www*?
d. Locate the directory */Users/Shared/unix1*. How v many files does it contain? How many directories?

Problem #2

Without using a text editor examine the contents of the file *cosmids1.txt*.

a. How many lines does this file contain?
b. How many characters?
c. What is the first line of this file?
d. What are the last 3 lines?

The files *cosmids1.txt*, *cosmids2.txt*, *cosmids3.txt*, and *cosmids4.txt* each contain lists of predicted genes from the *C. elegans* genome.

c. Using the **grep** program, find the file(s) that contains the gene ZK103.4.
d. Copy these four files into your home directory using *one* command only.
e. Rename *cosmids1.txt* to *clones.txt*.
f. Create a new subdirectory named *delete_me*. Move all the cosmids file into it.
g. Use the **chmod** command to make this new subdirectory and its contents read-only.
h. Now delete *delete_me* and all its contents using the recursive form of **rm**. What effect do the read-only file permissions have on this?

Problem #3

Create a text file using the **emacs**, or **acquamacs** text editor. Enter your name and address and save the file as *address.txt*.

What is the Command Line?

Underlying the pretty Mac OSX GUI is a powerful command-line operating system. The command line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Many bioinformatics tools are written to run on the command line and have no graphical interface. In many cases, a command line tool is more versatile than a graphical tool, because you can easily combine command line tools into automated scripts that accomplish tasks without human intervention.

In this course, we will be writing Perl scripts that are completely command-line based.

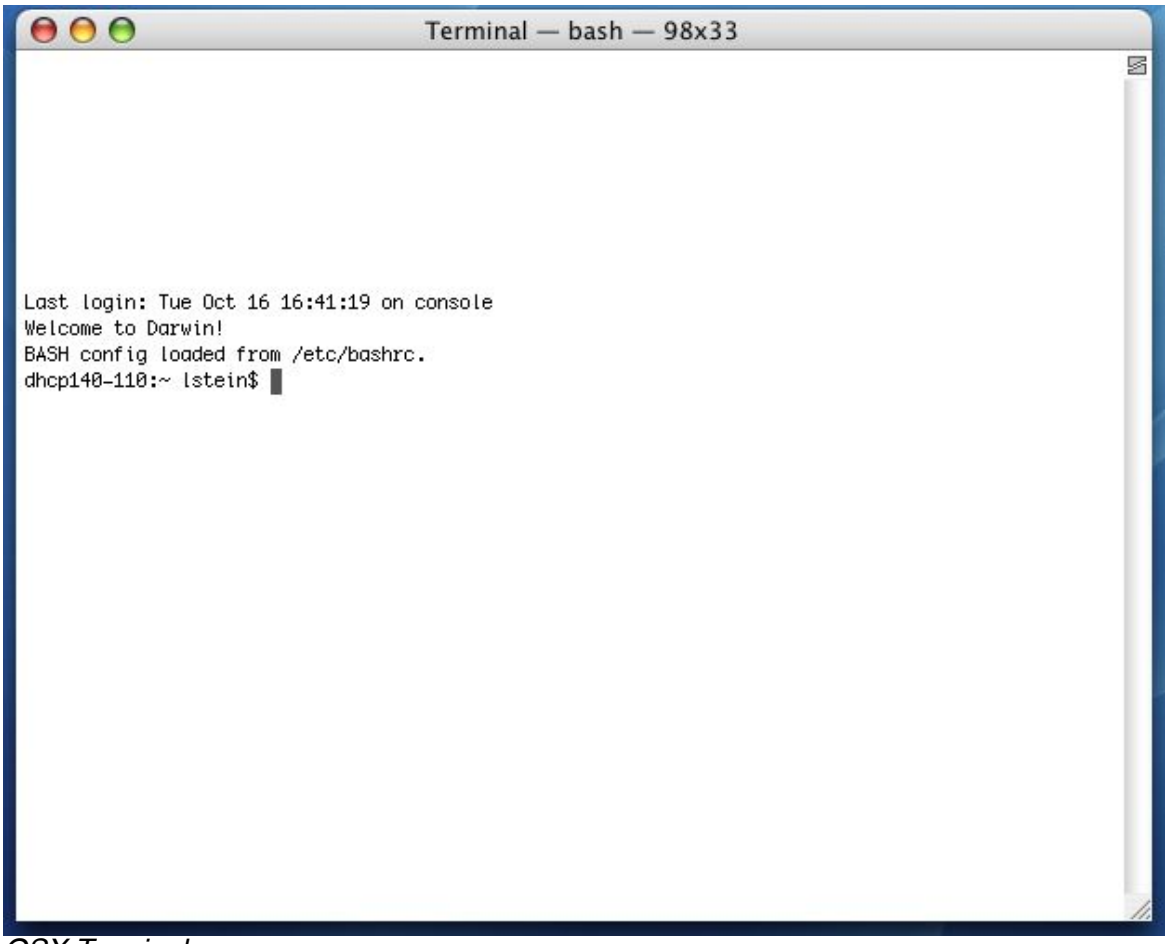
Logging into Your Workstation

Your workstation is an iMac. To log into it, provide the following information:

> *Your username:* the initial of your first name, followed by your full last name. For example, my username is **srobb** for **s**ofia **robb**
> *Your password:* **changeme**

Bringing up the Command Line

To bring up the command line, use the Finder to navigate to *Applications->Utilities* and double-click on the *Terminal* application. This will bring up a window like the following:



OSX Terminal

You will be using this application a lot, so I suggest that you drag the Terminal icon into the shortcuts bar at the bottom of your screen.

OK. I've Logged in. What Now?

The terminal window is running a **shell** called "bash." The shell is a loop that:

1. Prints a prompt
2. Reads a line of input from the keyboard
3. Parses the line into one or more commands
4. Executes the commands (which usually print some output to the terminal)
5. Prints the prompt
6. Repeat...

There are many different shells with bizarre names like **bash**, **sh**, **csh**, **tcsh**, **ksh**, and **zsh**. The "sh" part means shell. Each shell was designed for the purpose of confusing you and tripping you up. We have set up your accounts to use **bash**. Stay with **bash** and you'll get used to it, eventually.

Command-Line Prompt

Most of bioinformatics is done with command-line software, so you should take some time to learn to use the shell effectively.

This is a command line prompt:

```
bush202>
```

This is another:

```
(~) 51%
```

This is another:

```
lstein@bush202 1:12PM>
```

What you get depends on how the system administrator has customized your login. You can customize yourself when you know how.

The prompt tells you the shell is ready to accept a command. When a long-running command is going, the prompt will not reappear until the system is ready to deal with your next request.

Issuing Commands

Type in a command and press the <Enter> key. If the command has output, it will appear on the screen. Example:

```
(~) 53% ls -F
GNUstep/                cool_elegans.movies.txt  man/
INBOX                   docs/                    mtv/
INBOX~                  etc/                     nsmail/
Mail@                   games/                   pcod/
News/                   get_this_book.txt        projects/
axhome/                 jcod/                    public_html/
bin/                    lib/                     src/
build/                  linux/                   tmp/
ccod/
(~) 54%
```

The command here is *ls -F*, which produces a listing of files and directories in the current directory (more on which later). After its output, the command prompt appears agin.

Some programs will take a long time to run. After you issue their command name, you won't recover the shell prompt until they're done. You can either launch a new shell (from Terminal's File menu), or run the command in the background using the ampersand:

```
(~) 54% long_running_application&
(~) 55%
```

The command will now run in the background until it is finished. If it has any output, the output will be printed to the terminal window. You may wish to redirect the output as described later.

Command Line Editing

Most shells offer command line entering. Up until the comment you press <Enter>, you can go back

over the command line and edit it using the keyboard. Here are the most useful keystrokes:

Backspace
: Delete the previous character and back up one.

Left arrow, right arrow
: Move the text insertion point (cursor) one character to the left or right.

control-A (^A)
: Move the cursor to the beginning of the line. Mnemonic: A is first letter of alphabet

control-E (^E)
: Move the cursor to the end of the line. Mnemonic: <E> for the End (^Z was already taken for something else).

control-D (^D)
: Delete the character currently under the cursor. D=Delete.

control-K (^K)
: Delete the entire line from the cursor to the end. K=Kill. The line isn't actually deleted, but put into a temporary holding place called the "kill buffer".

control-Y (^Y)
: Paste the contents of the kill buffer onto the command line starting at the cursor. Y=Yank.

Up arrow, down arrow
: Move up and down in the command history. This lets you reissue previous commands, possibly after modifying them.

There are also some useful shell commands you can issue:

history
: Show all the commands that you have issued recently, nicely numbered.

!<number>
: Reissue an old command, based on its number (which you can get from *history*)

!!
: Reissue the immediate previous command.

!<partial command string>
: Reissue the previous command that began with the indicated letters. For example *!l* would reissue the *ls -F* command from the earlier example.

bash offers automatic command completion and spelling correction. If you type part of a command and then the tab key, it will prompt you with all the possible completions of the command. For example:

```
(~) 51% fd<tab>
(~) 51% fd
fd2ps    fdesign  fdformat fdlist   fdmount  fdmountd fdrawcmd fdumount
(~) 51%
```

If you hit tab after typing a command, but before pressing <Enter>, **bash** will prompt you with a list of file names. This is because many commands operate on files.

Wildcards

You can use wildcards when referring to files. "*" refers to zero or more characters. "?" refers to any single character. For example, to list all files with the extension ".txt", run **ls** with the pattern "*.txt":

```
(~) 56% ls -F *.txt
final_exam_questions.txt  genomics_problem.txt
genebridge.txt            mapping_run.txt
```

There are several more advanced types of wildcard patterns which you can read about in the **tcsh** manual page. For example, you can refer to files beginning with the characters "f" or "g" and ending with ".txt" this way:

```
(~) 57% ls -F [f-g]*.txt
final_exam_questions.txt  genebridge.txt            genomics_problem.tx
```

Home Sweet Home

When you first log in, you'll be placed in a part of the system that is your personal domain, called the *home directory*. You are free to do with this area what you will: in particular you can create and delete files and other directories. In general, you cannot create files elsewhere in the system.

Your home directory lives somewhere way down deep in the bowels of the system. On our iMacs, it is a directory with the same name as your login name, located in **/Users**. The full directory path is therefore **/Users/username**. Since this is a pain to write, the shell allows you to abbreviate it as *~username* (where "username" is your user name), or simply as ~. The weird character (technically called the "twiddle") is usually hidden at the upper left corner of your keyboard.

To see what is in your home directory, issue the command *ls -F*:

```
(~) % ls -F
INBOX          Mail/          News/          nsmail/        public_html/
```

This shows one file "INBOX" and four directories ("Mail", "News") and so on. (The "-F" in the command turns on fancy mode, which appends special characters to directory listings to tell you more about what you're seeing. "/" means directory.)

In addition to the files and directories shown with *ls -F*, there may be one or more hidden files. These are files and directories whose names start with a "." (technically called the "dot" character). To see these hidden files, add an "a" to the options sent to the *ls* command:

```
(~) % ls -aF
./                .cshrc            .login            Mail/
../               .fetchhost        .netscape/        News/
.Xauthority       .fvwmrc           .xinitrc*         nsmail/
.Xdefaults        .history          .xsession@        public_html/
.bash_profile     .less             .xsession-errors
.bashrc           .lessrc           INBOX
```

Whoa! There's a lot of hidden stuff there. But don't go deleting dot files willy-nilly. Many of them are esential configuration files for commands and other programs. For example, the *.profile* file contains configuration information for the **bash** shell. You can peek into it and see all of **bash**'s many options. You can edit it (when you know what you're doing) in order to change things like the command prompt and command search path.

Getting Around

You can move around from directory to directory using the *cd* command. Give the name of the directory you want to move to, or give no name to move back to your home directory. Use the *pwd*

command to see where you are (or rely on the prompt, if configured):

```
(~/docs/grad_course/i) 56% cd
(~) 57% cd /
(/) 58% ls -F
bin/          dosc/         gmon.out      mnt/          sbin/
boot/         etc/          home@         net/          tmp/
cdrom/        fastboot      lib/          proc/         usr/
dev/          floppy/       lost+found/   root/         var/
(/) 59% cd ~/docs/
(~/docs) 60% pwd
/usr/home/lstein/docs
(~/docs) 62% cd ../projects/
(~/projects) 63% ls
Ace-browser/                bass.patch
Ace-perl/                   cgi/
Foo/                        cgi3/
Interface/                  computertalk/
Net-Interface-0.02/         crypt-cbc.patch
Net-Interface-0.02.tar.gz   fixer/
Pts/                        fixer.tcsh
Pts.bak/                    introspect.pl*
PubMed/                     introspection.pm
SNPdb/                      rhmap/
Tie-DBI/                    sbox/
ace/                        sbox-1.00/
atir/                       sbox-1.00.tgz
bass-1.30a/                 zhmapper.tar.gz
bass-1.30a.tar.gz
(~/projects) 64%
```

Each directory contains two special hidden directories named "." and "..". "." refers always to the directory in which it is located. ".." refers always to the parent of the directory. This lets you move upward in the directory hierarchy like this:

```
(~/docs) 64% cd ..
```

and to do arbitrarily weird things like this:

```
(~/docs) 65% cd ../../docs
```

The latter command moves upward to levels, and then into a directory named "docs".

If you get lost, the *pwd* command prints out the full path to the current directory:

```
(~) 56% pwd
/Users/lstein
```

Essential Unix Commands

With the exception of a few commands that are built directly into the shell, all Unix commands are standalone executable programs. When you type the name of a command, the shell will search

through all the directories listed in the PATH environment variable for an executable of the same name. If found, the shell will execute the command. Otherwise, it will give a "command not found" error.

Most commands live in `/bin`, `/usr/bin`, or `/usr/local/bin`.

Getting Information About Commands

The **man** command will give a brief synopsis of the command:

```
(~) 76% man wc
Formatting page, please wait...
WC(1)                                                          WC(1)

NAME
       wc - print the number of bytes, words, and lines in files

SYNOPSIS
       wc [-clw] [--bytes] [--chars] [--lines] [--words] [--help]
       [--version] [file...]

DESCRIPTION
       This manual page documents the  GNU  version  of  wc.   wc
       counts  the  number  of bytes, whitespace-separated words,
...
```

Finding Out What Commands are There

The **apropos** command will search for commands matching a keyword or phrase:

```
(~) 100% apropos column
showtable (1)        - Show data in nicely formatted columns
colrm (1)            - remove columns from a file
column (1)           - columnate lists
fix132x43 (1)        - fix problems with certain (132 column) graphics
modes
```

Arguments and Command Switches

Many commands take arguments. Arguments are often (but not inevitably) the names of one or more files to operate on. Most commands also take command-line "switches" or "options" which fine-tune what the command does. Some commands recognize "short switches" that consist of a single character, while others recognize "long switches" consisting of whole words.

The **wc** (word count) program is an example of a command that recognizes both long and short options. You can pass it the **-c**, **-w** and/or **-l** options to count the characters, words and lines in a text file, respectively. Or you can use the longer but more readable, **--chars**, **--words** or **--lines** options. Both these examples count the number of characters and lines in the text file `/var/log/messages`:

```
(~) 102% wc -c -l /var/log/messages
     23     941 /var/log/messages
(~) 103% wc --chars --lines /var/log/messages
```

```
      23     941 /var/log/messages
```

You can cluster short switches by concatenating them together, as shown in this example:

```
(~) 104% wc -cl /var/log/messages
     23     941 /var/log/messages
```

Many commands will give a brief usage summary when you call them with the **-h** or **--help** switch.

Spaces and Funny Characters

The shell uses whitespace (spaces, tabs and other nonprinting characters) to separate arguments. If you want to embed whitespace in an argument, put single quotes around it. For example:

```
mail -s 'An important message' 'Lincoln Stein <lstein@cshl.org>'
```

This will send an e-mail to me. The **-s** switch takes an argument, which is the subject line for the e-mail. Because the desired subject contains spaces, it has to have quotes around it. Likewise, my e-mail address, which contains embedded spaces, must also be quoted in this way.

Certain special non-printing characters have *escape codes* associated with them:

Escape Code	Description
\n	new line character
\t	tab character
\r	carriage return character
\a	bell character (ding! ding!)
\nnn	the character whose ASCII code in octal is **nnn**

Useful Commands

Here are some commands that are used extremely frequently. Use **man** to learn more about them. Some of these commands may be useful for solving the problem set ;-)

Manipulating Directories

ls
: Directory listing. Most frequently used as **ls -F** (decorated listing) and **ls -l** (long listing).

mv
: Rename or move a file or directory.

cp
: Copy a file.

rm
: Remove (delete) a file.

mkdir
: Make a directory

rmdir
: Remove a directory

ln
: Create a symbolic or hard link.

chmod
Change the permissions of a file or directory.

Manipulating Files

cat
Concatenate program. Can be used to concatenate multiple files together into a single file, or, much more frequently, to send the contents of a file to the terminal for viewing.
more
Scroll through a file page by page. Very useful when viewing large files. Works even with files that are too big to be opened by a text editor.
less
A version of **more** with more features.
head
View the head (top) of a file. You can control how many lines to view.
tail
View the tail (bottom) of a file. You can control how many lines to view. You can also use **tail** to view a growing file.
wc
Count words, lines and/or characters in one or more files.
tr
Substitute one character for another. Also useful for deleting characters.
sort
Sort the lines in a file alphabetically or numerically.
uniq
Remove duplicated lines in a file.
cut
Remove sections from each line of a file or files.
fold
Wrap each input line to fit in a specified width.
grep
Filter a file for lines matching a specified pattern. Can also be reversed to print out lines that don't match the specified pattern.
gzip (gunzip)
Compress (uncompress) a file.
tar
Archive or unarchive an entire directory into a single file.
emacs
Run the Emacs text editor (good for experts).

Networking

telnet
Log into a remote host machine.
ssh
A secure (encrypted) version of **telnet**.
ping
See if a remote host is up.
ftp
Transfer files using the File Transfer Protocol.
who
See who else is logged in.
lp
Send a file or set of files to a printer.

Standard I/O and Command Redirection

Unix commands communicate via the command line interface. They can print information out to the terminal for you to see, and accept input from the keyboard (that is, from *you*!)

Every Unix program starts out with three connections to the outside world. These connections are called "streams" because they act like a stream of information (metaphorically speaking):

standard input
: This is a communications stream initially attached to the keyboard. When the program reads from standard input, it reads whatever text you type in.

standard output
: This stream is initially attached to the command window. Anything the program prints to this channel appears in your terminal window.

standard error
: This stream is also initially attached to the command window. It is a separate channel intended for printing error messages.

The word "initially" might lead you to think that standard input, output and error can somehow be detached from their starting places and reattached somewhere else. And you'd be right. You can attach one or more of these three streams to a file, a device, or even to another program. This sounds esoteric, but it is actually very useful.

A Simple Example

The **wc** program counts lines, characters and words in data sent to its standard input. You can use it interactively like this:

```
(~) 62% wc
Mary had a little lamb,
little lamb,
little lamb.

Mary had a little lamb,
whose fleece was white as snow.
^D
      6      20     107
```

In this example, I ran the **wc** program. It waited for me to type in a little poem. When I was done, I typed the END-OF-FILE character, control-D (^D for short). **wc** then printed out three numbers indicating the number of lines, words and characters in the input.

More often, you'll want to count the number of lines in a big file; say a file filled with DNA sequences. You can do this by *redirecting* **wc**'s standard input from a file. This uses the **<** metacharacter:

```
(~) 63% wc <big_file.fasta
      2943    2998     419272
```

If you wanted to record these counts for posterity, you could redirect standard output as well using the **>** metacharacter:

```
(~) 64% wc <big_file.fasta >count.txt
```

Now if you **cat** the file *count.txt*, you'll see that the data has been recorded. **cat** works by taking its standard input and copying it to standard output. We redirect standard input from the *count.txt* file, and leave standard output at its default, attached to the terminal:

```
(~) 65% cat <count.txt
      2943    2998     419272
```

Redirection Meta-Characters

Here's the complete list of redirection commands for **bash**:

<filename	Redirect standard input to file
>filename	Redirect standard output to file
1>*filename*	Redirect just standard output to file (same as above)
2>*filename*	Redirect just standard error to file
>*filename* 2>&1	Redirect both stdout and stderr to file

These can be combined. For example, this command redirects standard input from the file named */etc/passwd*, writes its results into the file *search.out*, and writes its error messages (if any) into a file named *search.err*. What does it do? It searches the password file for a user named "root" and returns all lines that refer to that user.

```
(~) 66% grep root </etc/passwd >search.out 2>search.err
```

Filters, Filenames and Standard Input

Many Unix commands act as filters, taking data from a file or standard input, transforming the data, and writing the results to standard output. Most filters are designed so that if they are called with one or more filenames on the command line, they will use those files as input. Otherwise they will act on standard input. For example, these two commands are equivalent:

```
(~) 66% grep 'gatttgc' <big_file.fasta
(~) 67% grep 'gatttgc' big_file.fasta
```

Both commands use the **grep** command to search for the string "gatttgc" in the file *big_file.fasta*. The first one searches standard input, which happens to be redirected from the file. The second command is explicitly given the name of the file on the command line.

Sometimes you want a filter to act on a series of files, one of which happens to be standard input. Many filters let you use "-" on the command line as an alias for standard input. Example:

```
(~) 68% grep 'gatttgc' big_file.fasta bigger_file.fasta -
```

This example searches for "gatttgc" in three places. First it looks in *big_file.fasta*, then in *bigger_file.fasta*, and lastly in standard input (which, since it isn't redirected, will come from the keyboard).

Standard I/O and Pipes

The coolest thing about the Unix shell is its ability to chain commands together into pipelines. Here's an example:

```
(~) 65% grep gatttgc big_file.fasta | wc -l
22
```

There are two commands here. **grep** searches a file or standard input for lines containing a particular string. Lines which contain the string are printed to standard output. **wc -l** is the familiar word count program, which counts words, lines and characters in a file or standard input. The **-l** command-line option instructs **wc** to print out just the line count. The **|** character, which is known as the "pipe" character, connects the two commands together so that the standard output of **grep** becomes the standard input of **wc**.

What does this pipe do? It prints out the number of lines in which the string "gatttgc" appears in the file *big_file.fasta*.

More Pipe Idioms

Pipes are very powerful. Here are some common command-line idioms.

Count the Number of Times a Pattern does NOT Appear in a File

The example at the top of this section showed you how to count the number of lines in which a particular string pattern appears in a file. What if you want to count the number of lines in which a pattern does **not** appear?

Simple. Reverse the test with the **grep -v** switch:

```
(~) 65% grep -v gatttgc big_file.fasta | wc -l
2921
```

Uniquify Lines in a File

If you have a long list of names in a text file, and you are concerned that there might be some duplicates, this will weed out the duplicates:

```
(~) 66% sort long_file.txt | uniq > unique.out
```

This works by sorting all the lines alphabetically and piping the result to the **uniq** program, which removes duplicate lines that occur together. The output is placed in a file named *unique.out*.

Concatenate Several Lists and Remove Duplicates

If you have several lists that might contain repeated entries among them, you can combine them into a single unique list by **cat**ing them together, then uniquifying them as before:

```
(~) 67% cat file1 file2 file3 file4 | sort | uniq
```

Count Unique Lines in a File

If you just want to know how many unique lines there are in the file, add a **wc** to the end of the pipe:

```
(~) 68% sort long_file.txt | uniq | wc -l
```

Page Through a Really Long Directory Listing

Pipe the output of **ls** to the **more** program, which shows a page at a time. If you have it, the **less** program is even better:

```
(~) 69% ls -l | more
```

Monitor a Rapidly Growing File for a Pattern

Pipe the output of **tail -f** (which monitors a growing file and prints out the new lines) to **grep**. For example, this will monitor the */var/log/syslog* file for the appearance of e-mails addressed to *mzhang*:

```
(~) 70% tail -f /var/log/syslog | grep mzhang
```

Perl Scripting 1

Expressions, Operators, Statements, Variables

Lincoln Stein

Suggested Reading

Chapters 1, 2 & 5 of *Learning Perl*.

Lecture Notes

Problems

1. Create a script called "add" script to sum two arguments:

```
% add 2 3
5
```

2. Modify this script so that it checks that both arguments are present:

```
% add 2
Please provide two numeric arguments.
```

3. Create a script called "now" to print the current time of day:

```
% now
It is now Sun Jun 6 16:35:40 1999
```

4. Create a script to produce the reverse complement of a sequence (hint, use the **reverse** and **tr///** functions:

```
% reversec GAGAGAGAGAGTTTTTTTTT
AAAAAAAAACTCTCTCTCTC
```

What is Perl?

Perl is a Programming Language

Written by Larry Wall in late 80's to process mail on Unix systems and since extended by a huge cast of characters. The name is said to stand for:

1. Pathologically Eclectic Rubbish Lister
2. Practical Extraction and Report Language

Perl Properties

1. Interpreted Language
2. "Object-Oriented"
3. Cross-platform
4. Forgiving
5. Great for text
6. Extensible, rich set of libraries
7. Popular for web pages
8. Extremely popular for bioinformatics

Other Languages Used in Bioinformatics

C, C++
: Compiled languages, hence very fast.
Used for computation (BLAST, FASTA, Phred, Phrap, ClustalW)
Not very forgiving.

Java
: Interpreted, fully object-oriented language.
Built into web browsers.
Supposed to be cross-platform, but not quite yet.

Python
: Interpreted, fully object-oriented language.
Rich set of libraries.
Elegant syntax.
Smaller user community than Java or Perl.

Some Simple Scripts

Here are some simple scripts to illustrate the "look" of a Perl program.

Print a Message to the Terminal

Code:

```
# file: message.pl
print "When that Aprill with his shoures soote\n";
print "The droghte of March ath perced to the roote,\n";
print "And bathed every veyne in swich licour\n";
print "Of which vertu engendered is the flour...\n";
```

Output:

```
(~) 502> perl message.pl
When that Aprill with his shoures soote
The droghte of March ath perced to the roote,
And bathed every veyne in swich licour
```

```
Of which vertu engendered is the flour...
```

Do Some Math

Code:

```
# file: math.pl
print "2 + 2 =", 2+2, "\n";
print "log(1e23)= ", log(1e23), "\n";
print "2 * sin(3.1414)= ", 2 * sin(3.1414), "\n";
```

Output:

```
(~) 51% perl math.pl
2 + 2 =4
log(1e23)= 52.9594571388631
2 * sin(3.1414)= 0.000385307177203065
```

Run a System Command

Code:

```
# file: system.pl
system "ls";
```

Output:

```
(~/docs/grad_course/perl) 52% perl math.pl
index.html          math.pl~            problem_set.html~   what_is_perl.h
index.html~         message.pl          simple.html         what_is_perl.h
math.pl             problem_set.html    simple.html~
```

Return the Time of Day

Code:

```
# file: time.pl
$time = localtime;
print "The time is now $time\n";
```

Output:

```
(~) 53% perl time.pl
The time is now Thu Sep 16 17:30:02 1999
```

Mechanics of Writing Perl Scripts

Some hints to help you get going.

Creating the Script

A Perl script is just a text file. Use any text (programmer's) editor.

By convention, Perl script files end with the extension *.pl*.

The Emacs text editor has a *Perl mode* that will auto-format your Perl scripts and highlight keywords. Perl mode will be activated automatically if you end the script name with **.pl**.

Running the Script

Option 1
Run the **perl** program from the command line, giving it the name of the script file to run.

```
(~) 50% perl time.pl
The time is now Thu Sep 16 18:09:28 1999
```

Option 2
Put the magic comment *#!/usr/bin/perl* at the top of the script.

```
#!/usr/bin/perl
# file: time.pl
$time = localtime;
print "The time is now $time\n";
```

Make the script executable with *chmod +x time.pl*:

```
(~) 51% chmod +x time.pl
```

Run the script as if it were a command:

```
(~) 52% ./time.pl
The time is now Thu Sep 16 18:12:13 1999
```

Note that you have to type "./time.pl" rather than "time.pl" because, by default, **bash** does not search the current directory for commands to execute. To avoid this, you can add the current directory (".") to your search PATH environment variable. To do this, create a file in your home directory named `.profile` and enter the following line in it:

```
export PATH=$PATH:.
```

The next time you log in, your path will contain the current directory and you can type "time.pl" directly.

Common Errors

Every script goes through a few iterations before you get it right. Here are some common errors:

Syntax Errors

Code:

```
#!/usr/bin/perl
# file: time.pl
time = localtime;
print "The time is now $time\n";
```

Output:

```
(~) 53% time.pl
Can't modify time in scalar assignment at time.pl line 3, near "localtime;
Execution of time.pl aborted due to compilation errors.
```

Runtime Errors

Code:

```
#!/usr/bin/perl
# file: math.pl

$six_of_one = 6;
$half_dozen = 12/2;
$result = $six_of_one/($half_dozen - $six_of_one);
print "The result is $result\n";
```

Output:

```
(~) 54% math.pl
Illegal division by zero at math.pl line 6.
```

Forgetting to Make the Script Executable

```
(~) 55% test.pl
test.pl: Permission denied.
```

Getting the Path to Perl Wrong on the #! line

Code:

```
#!/usr/local/bin/pearl
# file: time.pl
$time = localtime;
print "The time is now $time\n";
```

```
(~) 55% time.pl
time.pl: Command not found.
```

Useful Perl Command-Line Options

You can call Perl with a few command-line options to help catch errors:

-c
Perform a syntax check, but don't run.

-w
Turn on verbose warnings.

-d
Turn on the Perl debugger.

Usually you will invoke these from the command-line, as in *perl -cw time.pl* (syntax check *time.pl* with verbose warnings). You can also put them in the top line: *#!/usr/bin/perl -w*.

Perl Statements

A Perl script consists of a series of *statements* and *comments*. Each statement is a command that is recognized by the Perl interpreter and executed. Statements are terminated by the semicolon character (;). They are also usually separated by a newline character to enhance readability.

A *comment* begins with the # sign and can appear anywhere. Everything from the # to the end of the line is ignored by the Perl interpreter. Commonly used for human-readable notes.

Some Statements

```
$sum = 2 + 2; # this is a statement

$f = <STDIN>; $g = $f++;  # these are two statements

$g = $f
  /
  $sum;         # this is one statement, spread across 3 lines
```

The Perl interpreter will start at the top of the script and execute all the statements, in order from top to bottom, until it reaches the end of the script. This execution order can be modified by loops and control structures.

Blocks

It is common to group statements into *blocks* using curly braces. You can execute the entire block conditionally, or turn it into a *subroutine* that can be called from many different places.

Example blocks:

```
{  # block starts
  my $EcoRI = 'GAATTC';
```

```
  my $sequence = <STDIN>;
  print "Sequence contains an EcoRI site" if $sequence=~/$EcoRI/;
}  # block ends

my $sequence2 = <STDIN>;
if (length($sequence) < 100) {  # another block starts
  print "Sequence is too small. Throw it back\n";
  exit 0;
} # and ends

foreach $sequence (@sequences) {  # another block
  print "sequence length = ",length($sequence),"\n";
}
```

Literals

Literals are constant values that you embed directly in the program code. Perl supports both *string literals* and *numeric literals*.

String Literals

String literals are enclosed by single quotes (') or double quotes ("):

```
'The quality of mercy is not strained.';  # a single-quoted string
"The quality of mercy is not strained.";  # a double-quoted string
```

The difference between single and double-quoted strings is that variables and certain special escape codes are interpolated into double quoted strings, but not in single-quoted ones. Here are some escape codes:

\n	New line
\t	Tab
\r	Carriage return
\f	Form feed
\a	Ring bell
\040	Octal character (octal 040 is the space character)
\0x2a	Hexadecimal character (hex 2A is the "*" character)
\cA	Control character (This is the ^A character)
\u	Uppercase next character
\l	Lowercase next character

\U	Uppercase everything until \E
\L	Lowercase everything until \E
\Q	Quote non-word characters until \E
\E	End \U, \L or \Q operation

```
"Here goes\n\tnothing!";
  # evaluates to:
  # Here goes
  #     nothing!

'Here goes\n\tnothing!';
  # evaluates to:
  # Here goes\n\tnothing!

"Here goes \unothing!";
  # evaluates to:
  # Here goes Nothing!

"Here \Ugoes nothing\E";
  # evaluates to:
  # Here GOES NOTHING!

"Alert! \a\a\a";
  # evaluates to:
  # Alert! (ding! ding! ding!)
```

Putting backslashes in strings is a problem because they get interpreted as escape sequences. To inclue a literal backslash in a string, double it:

```
"My file is in C:\\Program  Files\\Accessories\\wordpad.exe";

  # evaluates to: C:\Program Files\Accessories\wordpad.exe
```

Put a backslash in front of a quote character in order to make the quote character part of the string:

```
"She cried \"Oh dear! The parakeet has flown the coop!\"";

  # evaluates to: She cried "Oh dear! The parakeet has flown the coop!"
```

Numeric Literals

You can refer to numeric values using integers, floating point numbers, scientific notation, hexadecimal notation, and octal. With some help from the Math::Complex module, you can refer to complex numbers as well:

```
123;       # an integer

1.23;      # a floating point number

-1.23;     # a negative floating point number

1_000_000; # you can use _ to improve readability

1.23E45;   # scientific notation

0x7b;      # hexadecimal notation  (decimal 123)

0173;      # octal notation (decimal 123)

use Math::Complex;  # bring in the Math::Complex module

12+3*i;    # complex number 12 + 3i
```

Backtick Strings

You can also enclose a string in backtics (`). This has the unusual property of executing whatever is inside the string as a Unix system command, and returning its output:

```
`ls -l`;
# evaluates to a string containing the output of running the
# ls -l command
```

Lists

The last type of literal that Perl recognizes is the *list*, which is multiple values strung together using the comma operator (,) and enclosed by parentheses. Lists are closely related to *arrays*, which we talk about later.

```
('one', 'two', 'three', 1, 2, 3, 4.2);
  # this is 7-member list contains a mixure of strings, integers
  # and floats
```

Operators

Perl has numerous *operators* (over 50 of them!) that perform operations on string and numberic values. Some operators will be familiar from algebra (like "+", to add two numbers together), while others are more esoteric (like the "." string concatenation operator).

Numeric & String Operators

The "." operator acts on strings. The "!" operator acts on strings and numbers. The rest act on numbers.

Operator	Description	Example	Result
.	String concatenate	'Teddy' . 'Bear'	TeddyBear
=	Assignment	$a = 'Teddy'	$a variable contains 'Teddy'
+	Addition	3+2	5
-	Subtraction	3-2	1
-	Negation	-2	-2
!	Not	!1	0
*****	Multiplication	3*2	6
/	Division	3/2	1.5
%	Modulus	3%2	1
******	Exponentiation	3**2	9
<FILEHANDLE>	File input	<STDIN>	Read a line of input from standard input
>>	Right bit shift	3>>2	0 (binary 11>>2=00)
<<	Left bit shift	3<<2	12 (binary 11<<2=1100)
|	Bitwise OR	3|2	3 (binary 11|10=11
&	Bitwise AND	3&2	2 (binary 11&10=10
^	Bitwise XOR	3^2	1 (binary 11^10=01

Operator Precedence

When you have an expression that contains several operators, they are evaluated in an order determined by their *precedence*. The precedence of the mathematical operators follows the rules of arithmetic. Others follow a precedence that usually does what you think they should do. If uncertain, use parentheses to force precedence:

```
2+3*4;    # evaluates to 14, multiplication has precedence over addition
(2+3)*4;  # evaluates to 20, parentheses force the precedence
```

Logical Operators

These operators compare strings or numbers, returning TRUE or FALSE:

Numeric Comparison		String Comparison	
3 == 2	equal to	**'Teddy' eq 'Bear'**	equal to
3 != 2	not equal to	**'Teddy' ne 'Bear'**	not equal to
3 < 2	less than	**'Teddy' lt 'Bear'**	less than

3 > 2	greater than	**'Teddy' gt 'Bear'**	greater than
3 <= 2	less or equal	**'Teddy' le 'Bear'**	less than or equal
3 >= 2	greater than or equal	**'Teddy' ge 'Bear'**	greater than or equal
3 <=> 2	compare	**'Teddy' cmp 'Bear'**	compare
		'Teddy' =~ /Bear/	pattern match

The **<=>** and **cmp** operators return:

- **-1** if the left side is less than the right side
- **0** if the left side equals the right side
- **+1** if the left side is greater than the right side

File Operators

Perl has special *file operators* that can be used to query the file system. These operators generally return TRUE or FALSE.

Example:

```
print "Is a directory!\n" if -d '/usr/home';
print "File exists!\n" if -e '/usr/home/lstein/test.txt';
print "File is plain text!\n" if -T '/usr/home/lstein/test.txt';
```

There are many of these operators. Here are some of the most useful ones:

-e filename	file exists
-r filename	file is readable
-w filename	file is writable
-x filename	file is executable
-z filename	file has zero size
-s filename	file has nonzero size (returns size)
-d filename	file is a directory
-T filename	file is a text file
-B filename	file is a binary file
-M filename	age of file in days since script launched
-A filename	same for access time

Functions

In addition to its operators, Perl has many *functions*. Functions have a human-readable name, such as **print** and take one or more arguments passed as a list. A function may return no value, a single value (AKA "scalar"), or a list (AKA "array"). You can enclose the argument list in parentheses, or leave the parentheses off.

A few examples:

```
  # The function is print.  Its argument is a string.
  # The effect is to print the string to the terminal.
print "The rain in Spain falls mainly on the plain.\n";

  # Same thing, with parentheses.
print("The rain in Spain falls mainly on the plain.\n");

  # You can pass a list to print.  It will print each argument.
  # This prints out "The rain in Spain falls 6 times in the plain."
print "The rain in Spain falls ",2*4-2," times in the plain.\n";

  # Same thing, but with parentheses.
print ("The rain in Spain falls ",2*4-2," times in the plain.\n");

  # The length function calculates the length of a string,
  # yielding 45.
length "The rain in Spain falls mainly on the plain.\n";

  # The split function splits a string based on a delimiter pattern
  # yielding the list ('The','rain in Spain','falls mainly','on the plain.')
split '/','The/rain in Spain/falls mainly/on the plain.';
```

Often Used Functions (alphabetic listing)

For specific information on a function, use **perldoc -f *function_name*** to get a concise summary.

abs	absolute value
chdir	change current directory
chmod	change permissions of file/directory
chomp	remove terminal newline from string variable
chop	remove last character from string variable
chown	change ownership of file/directory
close	close a file handle
closedir	close a directory handle
cos	cosine
defined	test whether variable is defined
delete	delete a key from a hash

die	exit with an error message
each	iterate through keys & values of a hash
eof	test a filehandle for end of file
eval	evaluate a string as a perl expression
exec	quit Perl and execute a system command
exists	test that a hash key exists
exit	exit from the Perl script
glob	expand a directory listing using shell wildcards
gmtime	current time in GMT
grep	filter an array for entries that meet a criterion
index	find location of a substring inside a larger string
int	throw away the fractional part of a floating point number
join	join an array together into a string
keys	return the keys of a hash
kill	send a signal to one or more processes
last	exit enclosing loop
lc	convert string to lowercase
lcfirst	lowercase first character of string
length	find length of string
local	temporarily replace the value of a global variable
localtime	return time in local timezone
log	natural logarithm
m//	pattern match operation
map	perform on operation on each member of array or list
mkdir	make a new directory
my	create a local variable
next	jump to the top of enclosing loop
open	open a file for reading or writing
opendir	open a directory for listing
pack	pack a list into a compact binary representation

package	create a new namespace for a module
pop	pop the last item off the end of an array
print	print to terminal or a file
printf	formatted print to a terminal or file
push	push a value onto the end of an array
q/STRING/	generalized single-quote operation
qq/STRING/	generalized double-quote operation
qx/STRING/	generalized backtick operation
qw/STRING/	turn a space-delimited string of words into a list
rand	random number generator
read	read binary data from a file
readdir	read the contents of a directory
readline	read a line from a text file
readlink	determine the target of a symbolic link
redo	restart a loop from the top
ref	return the type of a variable reference
rename	rename or move a file
require	load functions defined in a library file
return	return a value from a user-defined subroutine
reverse	reverse a string or list
rewinddir	rewind a directory handle to the beginning
rindex	find a substring in a larger string, from right to left
rmdir	remove a directory
s///	pattern substitution operation
scalar	force an expression to be treated as a scalar
seek	reposition a filehandle to an arbitrary point in a file
select	make a filehandle the default for output
shift	shift a value off the beginning of an array
sin	sine
sleep	put the script to sleep for a while

sort	sort an array or list by user-specified criteria
splice	insert/delete array items
split	split a string into pieces according to a pattern
sprintf	formatted string creation
sqrt	square root
stat	get information about a file
sub	define a subroutine
substr	extract a substring from a string
symlink	create a symbolic link
system	execute an operating system command, then return to Perl
tell	return the position of a filehandle within a file
tie	associate a variable with a database
time	return number of seconds since January 1, 1970
tr///	replace characters in a string
truncate	truncate a file (make it smaller)
uc	uppercase a string
ucfirst	uppercase first character of a string
umask	change file creation mask
undef	undefine (remove) a variable
unlink	delete a file
unpack	the reverse of pack
untie	the reverse of tie
unshift	move a value onto the beginning of an array
use	import variables and functions from a library module
values	return the values of a hash variable
wantarray	return true in an array context
warn	print a warning to standard error
write	formatted report generation

Creating Your Own Functions

You can define your own functions or redefine the built-in ones using the **sub** function. This is described

in more detail in the lesson on creating subroutines, which you'll be seeing soon..

Variables

A variable is a symbolic placeholder for a value, a lot like the variables in algebra. Perl has several built-in variable types:

Scalars: **$variable_name**
: A single-valued variable, always preceded by a $ sign.

Arrays: **@array_name**
: A multi-valued variable indexed by integer, preceded by an @ sign.

Hashes: **%hash_name**
: A multi-valued variable indexed by string, preceded by a % sign.

Filehandle: **FILEHANDLE_NAME**
: A file to read and/or write from. Filehandles have no special prefix, but are usually written in all uppercase.

We discuss arrays, hashes and filehandles later.

Scalar Variables

Scalar variables have names beginning with $. The name must begin with a letter or underscore, and can contain as many letters, numbers or underscores as you like. These are all valid scalars:

- $foo
- $The_Big_Bad_Wolf
- $R2D2
- $_____A23
- $Once_Upon_a_Midnight_Dreary_While_I_Pondered_Weak_and_Weary

You assign values to a scalar variable using the **=** operator (not to be confused with **==**, which is numeric comparison). You read from scalar variables by using them wherever a value would go.

A scalar variable can contain strings, floating point numbers, integers, and more esoteric things. You don't have to predeclare scalars. A scalar that once held a string can be reused to hold a number, and vice-versa:

Code:

```
$p = 'Potato';  # $p now holds the string "potato"
$bushels = 3;   # $bushels holds the value 3
$potatoes_per_bushel = 80;  # $potatoes_per_bushel contains 80;

$total_potatoes = $bushels * $potatoes_per_bushel;  # 240

print "I have $total_potatoes $p\n";
```

Output:

```
I have 240 Potato
```

Scalar Variable String Interpolation

The example above shows one of the interesting features of double-quoted strings. If you place a scalar variable inside a double quoted string, it will be interpolated into the string. With a single-quoted string, no interpolation occurs.

To prevent interpolation, place a backslash in front of the variable:

```
print "I have \$total_potatoes \$p\n";

# prints: I have $total_potatoes $p
```

Operations on Scalar Variables

You can use a scalar in any string or numeric expression like `$hypotenuse = sqrt($x**2 + $y**2)` or `$name = $first_name . ' ' . $last_name`. There are also numerous shortcuts that combine an operation with an assignment:

$a++
: Increment $a by one

$a--
: Decrement $a by one

$a += $b
: Modify $a by adding $b to it.

$a -= $b
: Modify $a by subtracting $b from it.

$a *= $b
: Modify $a by multiplying $b to it.

$a /= $b
: Modify $a by dividing it by $b.

$a .= $b
: Modify the **string** in $a by appending $b to it.

Example Code:

```
$potatoes_per_bushel = 80;  # $potatoes_per_bushel contains 80;

$p = 'one';
$p .= ' ';      # append a space
$p .= 'potato'; # append "potato"

$bushels = 3;
$bushels *= $potatoes_per_bushel; # multiply
```

```
print "From $p come $bushels.\n";
```

Output:

```
From one potato come 240.
```

String Functions that Come in Handy for Dealing with Sequences

Reverse the Contents of a String

```
$name          = 'My name is Lincoln';
$reversed_name = reverse $name;
print $reversed_name,"\n";
# prints "nlocniL si eman yM"
```

Translating one set of letters into another set

```
$name = 'My name is Lincoln';
# swap a->g and c->t
$name =~ tr/ac/gt/;
print $name,"\n";
# prints "My ngme is Lintoln"
```

Can you see how a combination of these two operators might be useful for computing the reverse complement?

Processing Command Line Arguments

When a Perl script is run, its command-line arguments (if any) are stored in an automatic array called **@ARGV**. You'll learn how to manipulate this array later. For now, just know that you can call the **shift** function repeatedly from the main part of the script to retrieve the command line arguments one by one.

Printing the Command Line Argument

Code:

```
#!/usr/bin/perl
# file: echo.pl

$argument = shift;
print "The first argument was $argument.\n";
```

Output:

```
(~) 50% chmod +x echo.pl
(~) 51% echo.pl tuna
The first argument was tuna.
(~) 52% echo.pl tuna fish
The first argument was tuna.
(~) 53% echo.pl 'tuna fish'
The first argument was tuna fish.
(~) 53% echo.pl
The first argument was.
```

Computing the Hypotenuse of a Right Triangle

Code:

```
#!/usr/bin/perl
# file: hypotense.pl

$x = shift;
$y = shift;
$x>0 and $y>0 or die "Must provide two positive numbers";

print "Hypotenuse=",sqrt($x**2+$y**2),"\n";
```

Output:

```
(~) 82% hypotenuse.pl
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 83% hypotenuse.pl 1
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 84% hypotenuse.pl 3 4
Hypotenuse=5
(~) 85% hypotenuse.pl 20 18
Hypotenuse=26.9072480941474
(~) 86% hypotenuse.pl -20 18
Must provide two positive numbers at hypotenuse.pl line 6.
```

Perl Scripting II

Conditionals, Logical operators, Loops, and File handles

Suzi Lewis

Genome Informatics

Control Structures

- Control structures allow you to tell Perl the order in which you want the interpreter to execute statements.
  - You can create alternative branches in which different sets of statements are executed depending on circumstances
  - You can create various types of repetitive loops.

Conditional Blocks

```
$i = 1;
$j = 2;
if ($i == $j) {    # Curly braces define a “block”
  print "i equals j\n";
  $i += 1;
}

unless ($i == $j) {
  print "i does not equal j\n";
  $i += 2;
}
print "\$i is $i\n";
```

Single line Conditionals

- You can also use the operators **`if`** and **`unless`** at the end of a single statement. This executes that one statement conditionally.

```
print "i equals j\n"   if $i == $j;
print "i is twice j\n" if $i == $j * 2;

print "i does not equal j\n" unless $i == $j;
```

If-Else Statements

- Use else blocks for either/or constructions.

```
if ($i == $j) {
    print "i equals j\n";
    $i += $j;
} else {
    print "i does not equal j\n";
    die "Operation aborted!";
}
```

- What does this print if $i=2 and $j=2?

If-Else Statements

- You can perform multiple tests in a series using elsif:

```
if ($i > 100) {
    print "i is too large\n";
} elsif ($i < 0) {
    print "i is too small\n";
} elsif ($i == 50) {
    print "i is too average\n";
} else {
    print "i is just right!\n";
}
```

- What does this print if $i=50? If $i=51?

Use == to Compare Two Numbers for Equality

```
$i =  4 == 4;       # TRUE
$i =  4 == 2 + 2;   # TRUE
$i =  4 == $j;      # depends on what $j is
```

- Do not confuse == with =
  - == is for numeric comparison.
  - = is for assignment.

Use != to Compare Two numbers for Non-Equality

```
$i =  4 != 4;       # FALSE
$i =  4 != 2 + 2;   # FALSE
$i =  4 != $j;   # depends on what $j is
```

Use > and < for "Greater than", "Less than"

```
$i =  4 > 3;       # TRUE

$i =  4 < 3;       # FALSE

$i =  4 > $j;      # depends on what $j is
```

Use >= and <= for "Greater than or Equal", "Less than or Equal"

```
$i =  4 >= 3;       # TRUE

$i =  4 >= 4;       # TRUE

$i =  4 <= $j;    # depends on what $j is
```

Use <=> to Compare Two Numbers

```
$result = $i <=> $j
```

- $result is
  - -1 if the left side is less than the right side
  - 0 if the left side equals the right side
  - +1 if the left side is greater than the right side

- Nota Bene: The <=> operator is really useful in conjunction with the sort() function.

Use eq to Compare Two Strings for Equality

```
$i =  'fred' eq 'fred';       # TRUE
$i =  'fred and lucy' eq 'fred'.'and'.'lucy';  # TRUE
$i =  'fred' eq $j;       # depends on what $j is
```

- Do not confuse == with eq
  - == is for numeric comparison.
  - eq is for string comparison.

```
$i =  'fred' == 'lucy';       # WRONG WRONG WRONG!
```

Use ne to Compare Two Strings for Non-Equality

```
$i =  'fred' ne 'fred';       # FALSE

$i =  'fred' ne 'lucy';       # TRUE

$i =  'fred' eq $j # depends on what $j
  is
```

Use `gt`, `lt`, `ge`, `ne` for "Greater than", "Less than", "Greater or Equal" etc.

- String comparison is in ASCII alphabetic order.

- `$i = 'fred' gt 'lucy'; # FALSE`
- `$i = 'fred' lt 'lucy'; # TRUE`
- `$i = 'Lucy' lt 'lucy'; # TRUE`
- `$i = 'Lucy' lt 'fred'; # TRUE !!`

- In ASCII alphabetic order, the set of capital letters is less than the set of lowercase letters.

Use cmp to Compare Two Strings

```
$result = $i cmp $j
```

- $result is
  - -1 if the left side is less than the right side
  - 0 if the left side equals the right side
  - +1 if the left side is greater than the right side

- Nota Bene: cmp is also really useful in the sort() function.

What is TRUE (in Perl)

1. The string "0" is False.
2. The number 0 is False.
3. The empty string ("" or '') is False
4. The empty list is False
5. The undefined value is False (e.g. an uninitialized variable)
6. A number is false if it converts to string "0".
7. Everything else is True.

What is TRUE (in Perl)

- `$a;` # FALSE (not yet defined)
- `$a = 1;` # TRUE
- `$b = 0;` # FALSE
- `$c = "";` # FALSE
- `$d = 'true';` # TRUE
- `$e = 'false';` # TRUE (watch out! "false" is a non-empty string)
- `$f = ' ';` # TRUE (a single space is non-empty)
- `$g = "\n";` # TRUE (a single newline is non-empty)
- `@h = ();` # FALSE array is empty
- `$i = 0.0;` # FALSE
- `$j = '0.0';` # TRUE (watch out! The string "0.0" is not the same as "0")

Truth and the Comparison Operations

- If a comparison operation is true, it returns 1.
- If a comparison operation is false, it returns undefined.

```
$i =  4 == 1+3;
print "The answer is $i","\n";
```

- The answer is 1.

Logical Operators

- To combine comparisons, use the **and**, **or** and **not** logical operators.
  - **and** also known as &&,
  - **or** also known as ||
  - **not** also known as !
  - The short forms have higher precedence (will be interpreted first by Perl)

```
$i && $j   TRUE if $i AND $j are TRUE

$i || $j   TRUE if either $i OR $j are TRUE

!$i TRUE if $i is FALSE
```

Logical Operators Examples

```
if ($i < 100 && $i > 0) {
   print "a is the right size\n";
 } else {
   die "out of bounds error, operation aborted!";
 }

if ($i >= 100 || $i <= 0) {
     die "out of bounds error, operation aborted!";
}
```

To Reverse Truth, use not or !

```
$ok = ($i < 100 and $i > 0);
print "a is too small\n" if not $ok;

# same as this:
print "a is too small\n" unless $ok;

# and this:
print "a is too small\n" if !$ok;
```

`and` versus `&&`, `or` versus `||`

- Precedence
  - `&&` higher than `=` which is higher than `and`.
  - `||` higher than `=` which is higher than `or`.
- This is an issue in assignments
- Example 1

```
$ok = $i < 100 and $i > 0;
# This doesn't mean:
$ok = ($i < 100 and $i > 0);
# but:
($ok = $i < 100) and $i > 0;
```

- Example 2

```
$ok = $i < 100 && $i > 0;
# This does mean
$ok = ($i < 100 && $i > 0);
```

- When in doubt, use parentheses.

The or and || operators do no more than necessary.

- If what is on the left is true, then what is on the right is never evaluated, because it doesn't need to be.

```
$i = 10;
$j = 99;
# $j comparison never evaluated
$i < 100 or $j < 100;
```

The "or die" Idiom

- The die() function aborts execution with an error message
- You Combine "or die" and truth statements idiomatically like this

```
($i < 100 and $i > 0) or die "\$i is the wrong size";
```

File Tests

- A set of operators are used to check whether files exist, directories exist, files are readable, etc.
- -e <filename> # file exists
- -r <filename> # file is readable
- -x <filename> # file is executable
- -w <filename> # file is writable
- -d <filename> # filename is a directory
- Examples

```
(-w "./fasta.out")or die "Can't write to file";

print "This file is executable\n" if -x
"/usr/bin/perl";
```

Loops

- Loops let you execute the same piece of code over and over again.
- A `while` loop
  - Has a condition at the top.
  - The code within the block will execute until the condition becomes false.

```
while ( CONDITION ) {
    # Code to execute
}
```

While loop: Print "below 5" until the number is not below 5

- Code:

```
#!/usr/bin/perl

# file: counter.pl

$number = 0;

while ( $number < 5 ) {

   print "$number is less
   than 5\n";

   $number = $number + 1;

}
```

- Output of: 51% counter.pl

```
0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5
```

foreach Loops

- foreach will process each element of an array or list:

```
foreach $list_item ( @array ) {
    Do something with $list_item;
}
```

for Loops

- The for loop is the most general form of loop:

```
for ( initialization; test; update ) {
    # Do something
}
```

- The first time the loop is entered, the code at `initialization` is executed.
- Each time through the loop, the `test` is reevaluated and the loop stops if it returns false.
- After the execution of each loop, the code at `update` is performed.

A simple for loop

```
for ( $i = 1; $i < 5; $i++ ) {
    print $i,"\n";
}
```

- This will print out the numbers 1 to 4:
  - From command line: 52% loop.pl

```
1
2
3
4
```

- This is equivalent to the previous `while` example.
- There are ways to leave loops outside of the conditional, but this practice is discouraged.

Basic Input & Output (I/O)

Getting computer programs to talk to the rest of the world.

The STDIN, STDOUT and STDERR File handles

- Every Perl scripts starts out with three connections to the outside world:
- STDIN (Standard input)
  - Used to read input. By default connected to the keyboard, but can be changed from shell using redirection (<) or pipe (|).
- STDOUT (Standard output)
  - Used to write data out. By default connected to the terminal, but can be redirected to a file or other program from the shell using redirection or pipes.
- STDERR (Standard error)
  - Intended for diagnostic messages. By default connected to the terminal, etc.
- In addition to these 3 file handles, you can *create your own*.

Reading Data from STDIN

- To read a line of data into your program use the angle bracket function:

```
$line = <STDIN>
```

- <STDIN> will return one line of input as the result.
  - You usually will assign the result to a scalar variable.
  - The newline character is not removed line automatically; you have to do that yourself with chomp:

Reading Data from STDIN

```
print "Type your name: ";
$name = <STDIN>;
chomp $name;
if ($name eq 'Jim Watson') {
      print "Hail great master!;
else {
      print "Hello $name\n";
}
```

- The read/chomp sequence is often abbreviated as:

```
chomp($name = <STDIN>);
```

The Input Loop

- At the "end of file" (or when the user presses ^D to end input) <STDIN> will return whatever's left, which may or may not include a newline. Thereafter, <STDIN> will return an undefined value.
- This truthiness leads to typical input loop:

```
while ( $line = <STDIN> ) {
   chomp $line;
   # now do something with $line...
}
```

- This while loop will read one line of text after another. At the end of input, the <STDIN> function returns **undef** and the while loop terminates. Remember that even blank lines are TRUE, because they consist of a single newline character.

Output

- The print function writes data to output. In its full form, it takes a file handle as its first argument, followed by a list of scalars to print:
  - `print FILEHANDLE $data1,$data2,$data3,...`
- Notice there is no comma between FILEHANDLE and the data arguments.

- If FILEHANDLE is omitted it defaults to STDOUT. So these alternate statements are equivalent:

```
print STDOUT "Hello world\n";
print "Hello world\n";
```

- To print to standard error:

```
print STDERR "Does not compute.\n";
```

File handles

- You can create your own file handles using the `open` function
  - read and/or write to them
  - clean them up using `close`.
- `Open` prepares a file for reading and/or writing, and associates a file handle with it.
  - You can choose any name for the file handle, but the convention is to make it all caps. In the examples, we use FILEHANDLE.

Opening files for different purposes

- For reading

```
open FILEHANDLE,"cosmids.fasta"
```

  - alternative form:

```
open FILEHANDLE,"<cosmids.fasta"
```

- For writing

```
open FILEHANDLE,">cosmids.fasta"
```

- For appending

```
open FILEHANDLE,">>cosmids.fasta"
```

- For reading and writing

```
open FILEHANDLE,"+<cosmids.fasta"
```

Catching Open Failures

- It's very common for open to fail.
  - Maybe the file doesn't exist
  - you don't have permissions to read or create it.
- Always check open's return value, which is TRUE if the operation succeeded, FALSE otherwise:

```
$result = open COSMIDS,"cosmids.fasta";

die "Can't open cosmids file: $!\n" unless $result;
```

- When an error occurs, the $! variable holds a descriptive string containing a description of the error, such as "file not found".
- The compact idiom for accomplishing this in one step is:

```
open COSMIDS,"cosmids.fasta" or die "Can't open
  cosmids file: $!\n";
```

Using a File handle

- Once you've created any file handle, you can read from it or write to it, just as if it were STDIN or STDOUT.
- This code reads from file "text.in" and copies lines to "text.out":

```
open IN,"text.in" or die "Can't open input file: $!\n";
open OUT,">text.out" or die "Can't open output file: $!\n";
while ($line = <IN>) {
  print OUT $line;
}
```

- Here is a more compact way to do the same thing:

```
while (<IN>) {
  print OUT;
}
```

- And the minimalist solution:

```
print OUT while <IN>;
```

Closing a File handle

- When you are done with a filehandle, you should close it.
  - This will also happen automatically when your program ends
  - Or if you reuse the same file handle name.

```
close IN or warn "Errors closing filehandle: $!";
```

- Some errors, like file system full, only occur when you close the file handle, so *check* for errors in the same way you do when you open a file handle.

Pipes

- You can open pipes to and from other programs using the pipe ("|") symbol:

```
open PIPEIN, "ls -l |"  or die "Can't open pipe in: $!\n";
open PIPEOUT,"| sort"  or die "Can't open pipe out: $!\n";
print PIPEOUT while <PIPEIN>;
```

- This mysterious, silly example
  - Runs the ls -l command,
  - Reads its output a line at a time,
  - and does nothing but send the lines to the sort command.

More useful pipe example

- Count the # of occurrences of the string pBR322 in a file.

```
#!/usr/bin/perl
my $file = shift or die "Please provide a file name";
 if ($file =~ /\.gz$/) { # a GZIP file
    open IN,"gunzip -c $file |" or die "Can't open zcat pipe:
    $!\n";
 } else {
    open IN,$file or die "Can't open file: $!\n";
 }
 $count = 0;
 while (my $line = <IN>) {
    chomp $line;
    $count++ if $line eq 'pBR322';
 }
 close IN or die "Error while closing: $!\n";
 print "Found $count instances\n";
```

The Magic of <>

- The bare <> function when used without any explicit file handle is magical.
  - It reads from each of the files on the command line as if they were one single large file.
  - If no file is given on the command line, then <> reads from standard input.
- This can be extremely useful.

A Practical Example of <>

- Count the number of lines and bytes in a series of files. If no file is specified, count from standard input (like wc does).

```
#!/usr/bin/perl
($bytes,$lines) = (0,0);
while (my $line = <>) {
   $bytes += length($line);
   $lines++;
}
print "LINES: $lines\n";
print "BYTES: $bytes\n";
```

- Because <> uses open internally, you can use "-" to indicate standard input, or the pipe symbol to indicate a command pipe to create.

Perl Hygeine

- Because you don't have to predeclare variables in Perl, there is a big problem with typos:

```
$value = 42;
print "Value is OK\n" if $valu < 100; # UH OH
```

Use Warnings Will Warn of Uninitialized Variables

```
#!/usr/bin/perl

use warnings;

$value = 42;

print "Value is OK\n" if $valu < 100; # UH OH
```

- When run from the command line (% perl uninit.pl)

```
Name "main::valu" used only once: possible typo at
  uninit.pl line 4.

Name "main::value" used only once: possible typo at
  uninit.pl line 3.

Use of uninitialized value in numeric gt (>) at
  uninit.pl line 4.
```

"use strict"

- The "`use strict`" pragma forces you to predeclare all variables using "`my`":

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

use strict;

$value = 42;

print "Value is OK\n" if $valu < 100; # UH OH
```

- When run from command line (% perl uninit.pl)

```
Global symbol "$value" requires explicit package
  name at uninit.pl line 4.

Global symbol "$valu" requires explicit package
  name at uninit.pl line 5.

Execution of uninit.pl aborted due to compilation
  errors.
```

Using my

- Put a my in front of the variable in order to declare it:

```
#!/usr/bin/perl -w
use strict;
my $value = 42;
print "Value is OK\n" if $value < 100;
```

- When run from the command line (% perl uninit.pl)

```
Value is OK
```

- You can use "my" on a single variable, or on a list of variables. The only rule is to declare it before you use it.

```
my $value = 42;
my $a;
$a = 9;
my ($c,$d,$e,$f);
my ($first,$second) = (1,2);
```

Take home messages

ALWAYS use warnings

ALWAYS use strict

References

- Perl docs online perldoc.perl.org
- Learning Perl. Schwartz and Christiansen
  - Chapter 2

Perl Scripting III

Arrays and Hashes

(Also known as

Data Structures)

Ed Lee & Suzi Lewis

Genome Informatics

Basic Syntax

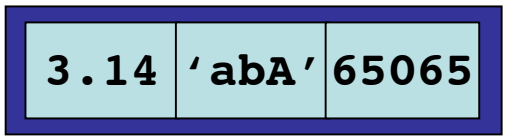
- In Perl the first character of the variable name determines how that variable will be interpreted when the code is run.
  - "$" indicates a "scalar" variable
  - "@" indicates an "array" variable
  - "%" indicates a "hash" variable
- You can have three variables with the same name
  - For example `$x`, `@x`, and `%x`
  - These represent three different things

An *Array* Is a *List* of Values

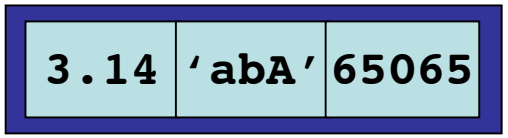
- For example, consider a list such as this
  - the number 3.14 as the first element
  - the string 'abA' as the second element
  - the number 65065 as the third element.
- How do you express this list in Perl?

"Literal Representation"

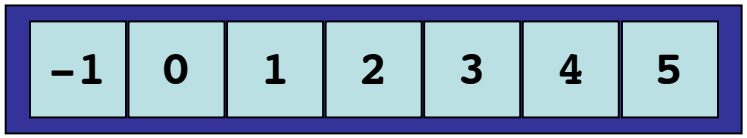
- Most simply
  - `my @array = (3.14, 'abA', 65065);`



- Or we can initialize from variables
  - `my $pi = 3.14;`
  - `my $s = 'abA';`
  - `my @array = ($pi, $s, 65065);`



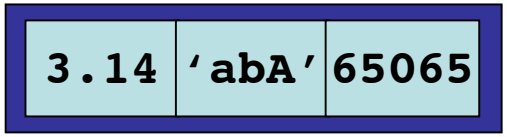
- We can also do integer ranges
  - `my @array = (-1..5); # shorthand for`

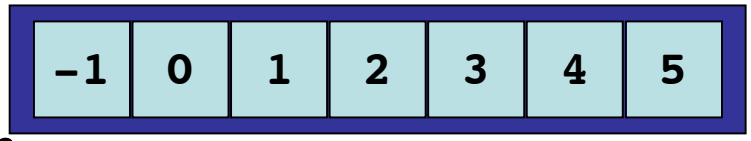


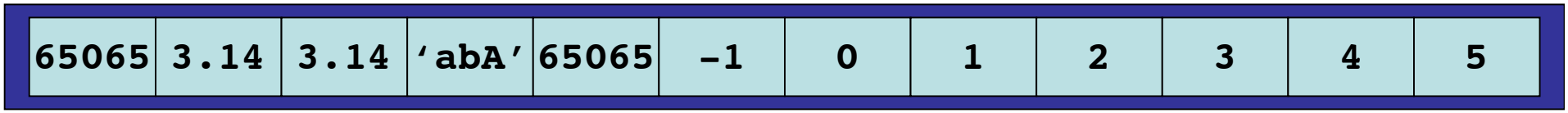
  - Counting down not allowed!

Array Variables and Assignment

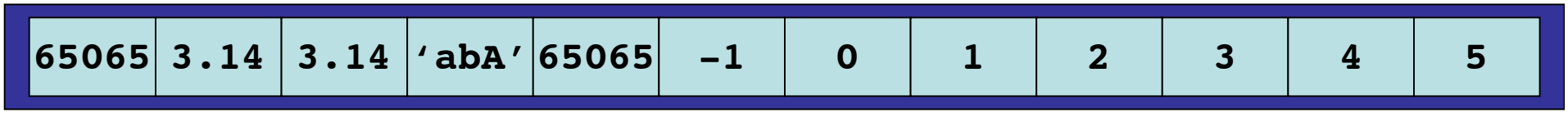
- `my $pi = 3.14;`
- `my $x = 65065;`
- `my @x = ($pi, 'abA', $x);`
- `my @y = (-1..5);`
- `my @z = ($x, $pi, @x, @y);`



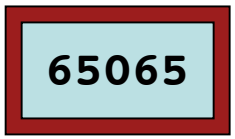


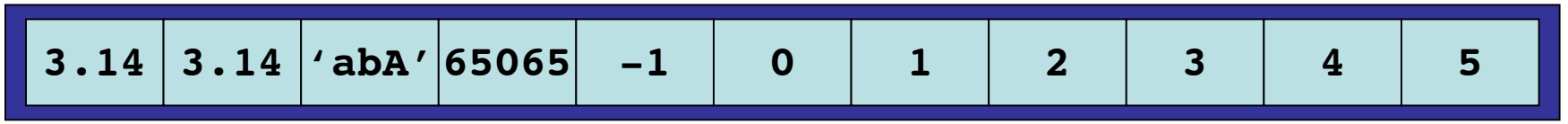


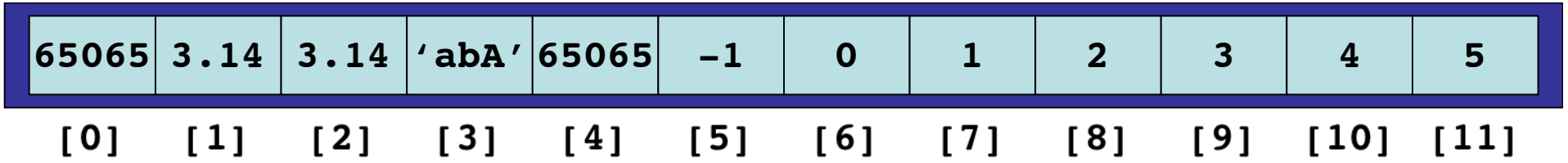
Array Variables and Assignment



- `my ($first, @rest) = @z;`

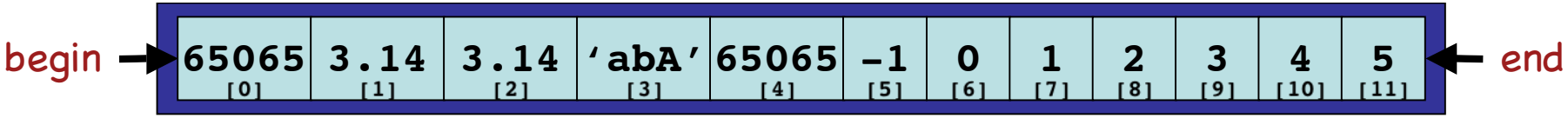






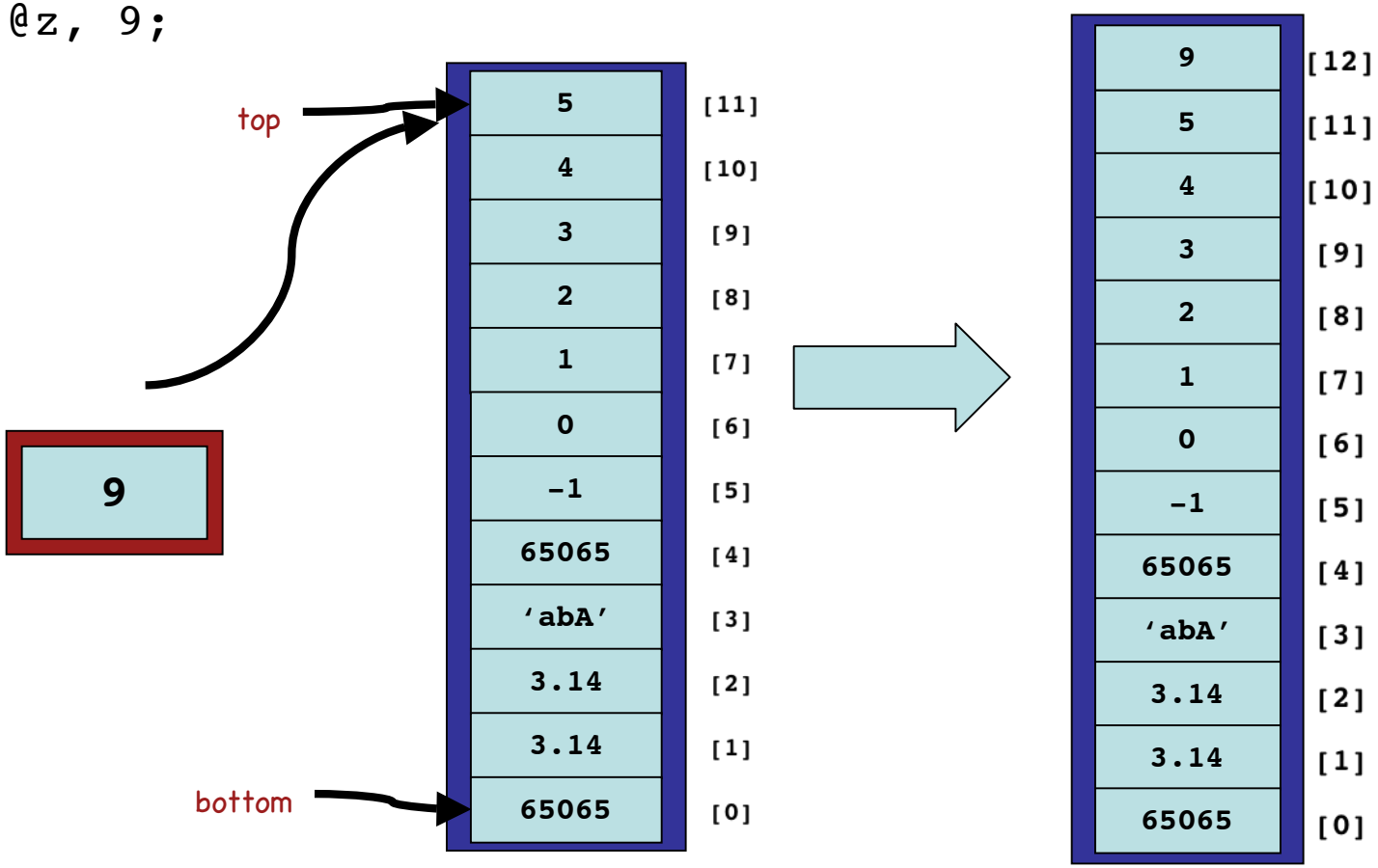
Getting at Array Elements

- `my $first = $z[0];` # 65065
- `$z[0] = 2; # assign a new value to the 1st item`
- `$first = $z[0];` # 2
- `my $max_index = $#z; # 11`
- `my $last = $z[$#z];` # 5



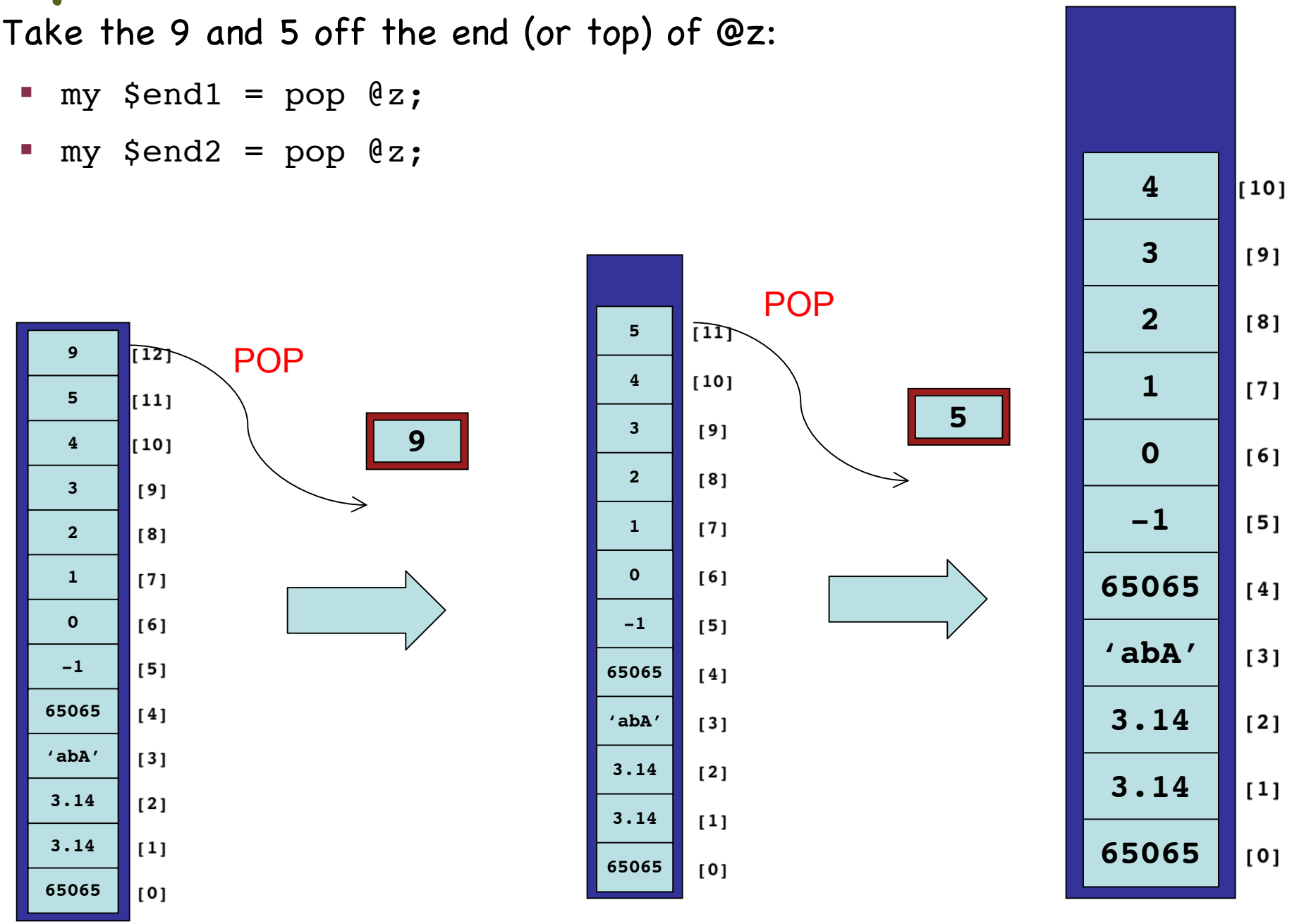
Push

- Add 9 to the end (or top) of @z;
  - `push @z, 9;`



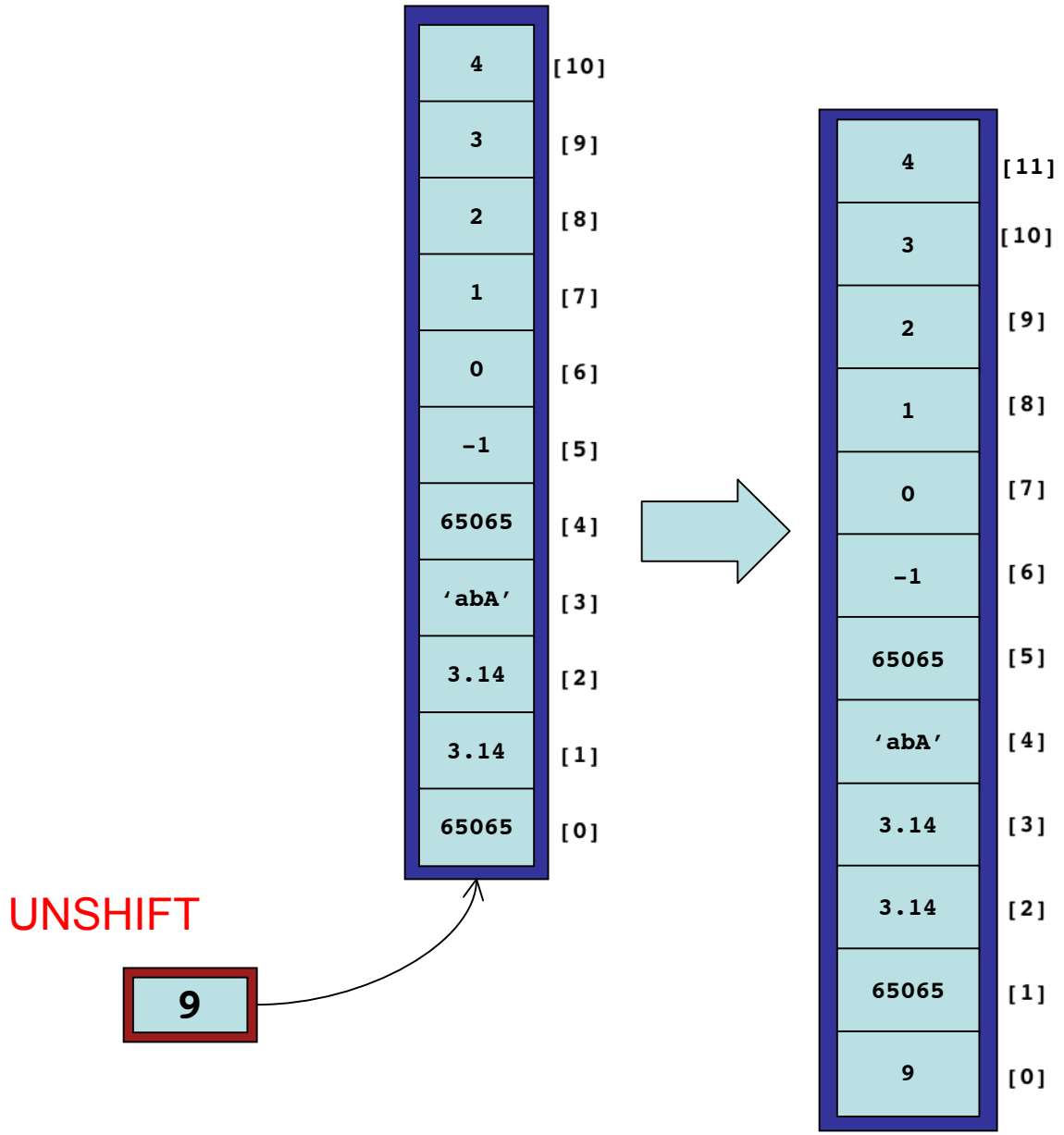
Pop

- Take the 9 and 5 off the end (or top) of @z:
  - `my $end1 = pop @z;`
  - `my $end2 = pop @z;`



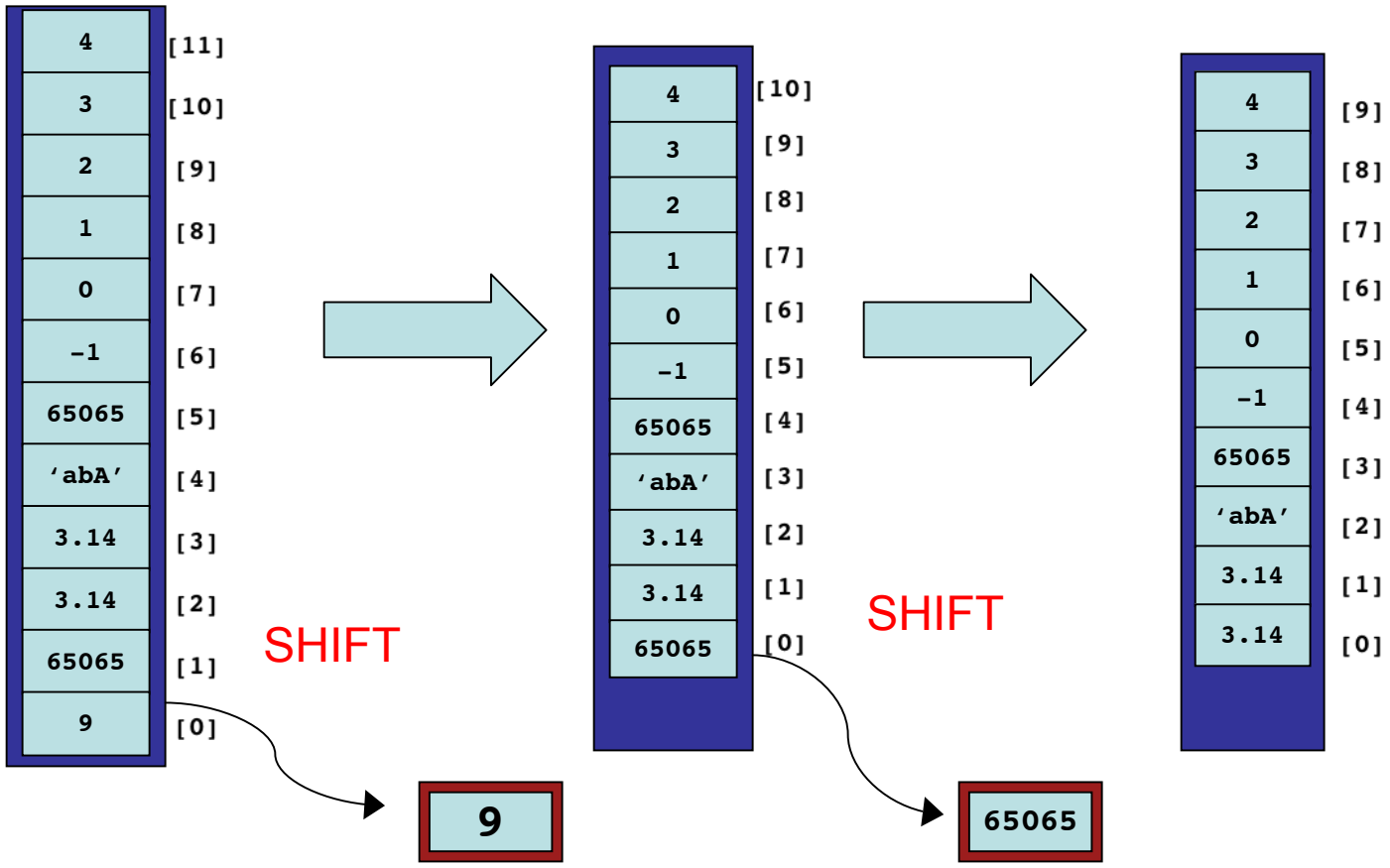
Unshift

- Add 9 to the beginning (or bottom) of @z;
  - `unshift @z, 9;`



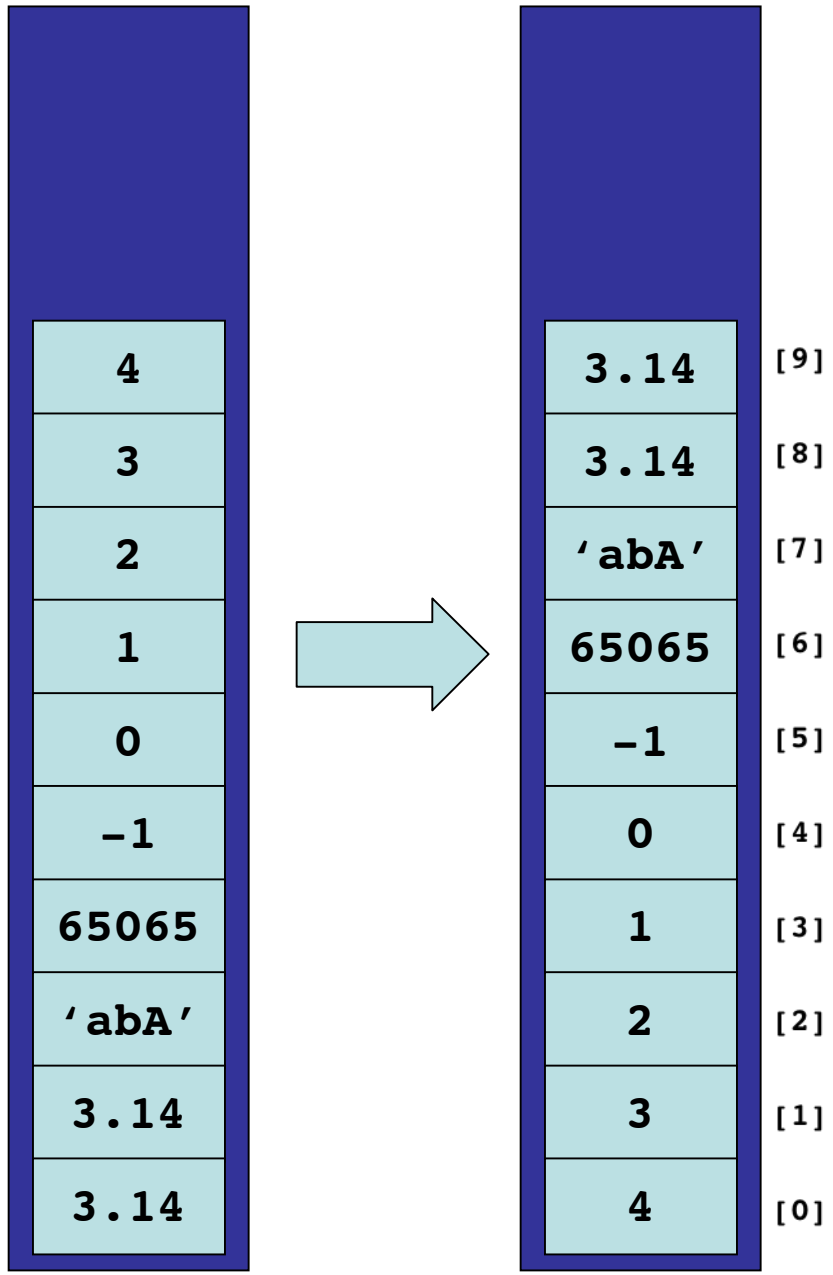
Shift

- Take 9 and then 65065 off the beginning of @z:
  - `my $b1 = shift @z;`
  - `my $b2 = shift @z;`



Reverse

- `my @zr = reverse @z;`



Array and Scalar Context

- The notion of array and scalar context is unique to Perl. Usually you can remain unaware of it, but it comes up in the **reverse** function.

```
print reverse 'abc';
```

```
abc
```

```
print reverse 'abc', 'def' , 'ghi' ;
```

```
ghidefabc
```

```
print scalar reverse 'abc';
```

```
cba
```

```
my $ba = reverse 'abc';
print $ba;
```

```
cba
```

Array and Scalar Context

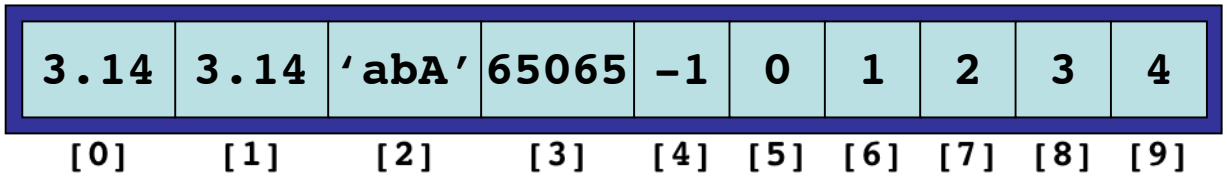
- The notion of array and scalar context can also be used to get the size of an array.

```
my @z = (1,2,3,4,5,6,7);
print scalar @z ," the number of elements in the
   array\n";
print $#z, ' this is max offset into scalar @z' , "\n";
```

7 the number of elements in the array

6 this is max offset into scalar @z

Iterating Through Array Contents



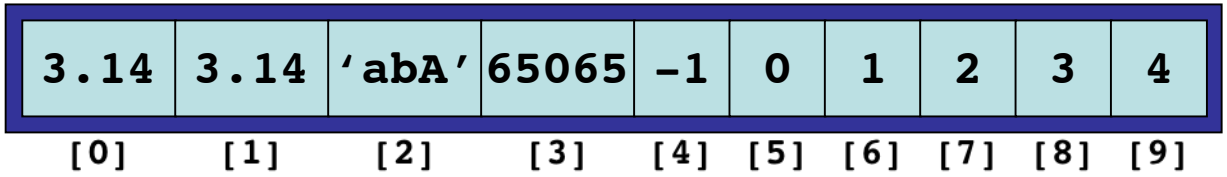
- Using a "foreach" loop

```
foreach my $array_value (@z) {
    print "$array_value\n";
}
```

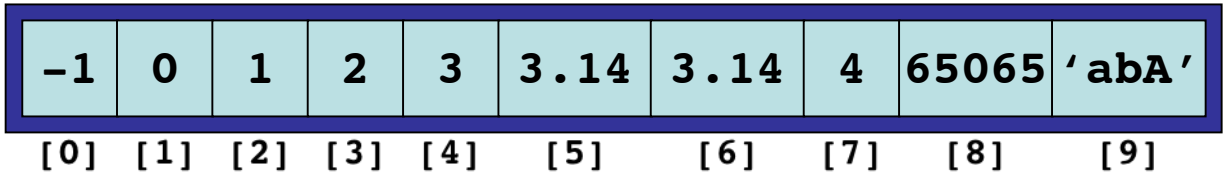
- Using a "for" loop

```
for (my $index = 0; $index < scalar(@z); ++$index) {
    my $array_value = $z[$index];
    print "$array_value\n";
}
```

Sorting



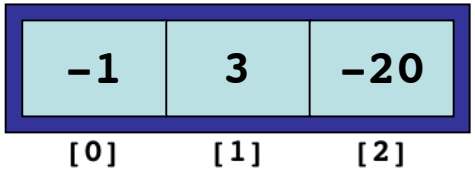
- Alphabetically:
  - `my @sortedArray = sort @z;`



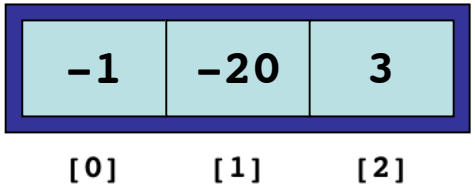
- This does exactly the same alphabetical sort
  - @sortedArray = sort {$a cmp $b} @z;

Sorting

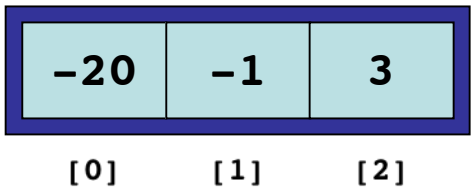
- An alphabetical sort (with only numbers in the array)
  - `my @numberArray = (-1, 3, -20);`



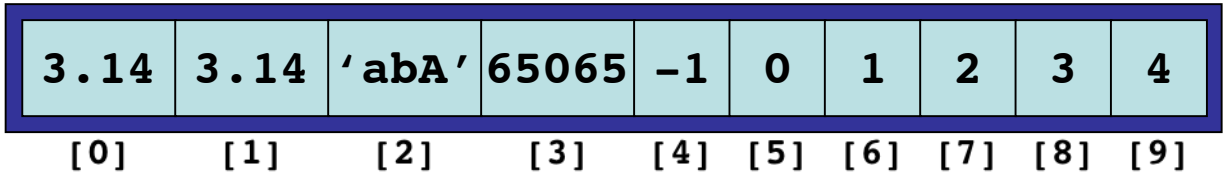
  - `my @sortedNums = sort @numberArray;`



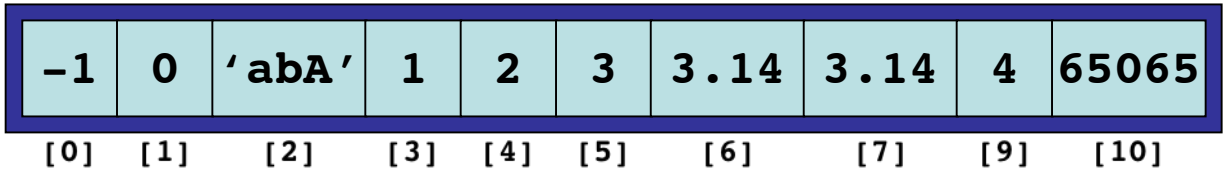
- Need a numerical sort to sort as numbers
  - my @sortedNums = sort {$a <=> $b} @numberArray;



Sorting



- What happens :
  - `@sortedArray = sort {$a <=> $b} @z;`
  - Argument "abA" isn't numeric in sort at arraySort.pl line 19.



Split and Join

- Split using a literal

```
my $string = "one,two,three";
my @array = split "," , $string;
print "@array" , ” - from array\n";

one two three - from array
```

- Join it up again

```
$string = join ‘:’, @array;
print $string , " - rejoined with colons\n";

one two three - from array
one:two:three - rejoined with colons
```

Split and Join

- Split using a regular expression

```
my $string = "one1two22three333fin";
my @array = split /\d+/ , $string;
print "@array" , "\n";

one two three fin
```

Swallowing Whole Files in a Single Gulp

- Read the file from stdin
  - `my @file = <>;`
- Eliminate newlines from each line
  - `chomp @file;`

A Hash Is a Lookup Table

- Hashes use a key to find an associated value.

```
my %translate; # the percent sign denotes a hash
$translate{'atg'} = 'M'; # codon is the key
$translate{'taa'} = '*'; # aa is the value
$translate{'ctt'} = 'K';  # lysine, oops
$translate{'ctt'} = 'L';  # leucine, fixed
print $translate{'atg'};
```

M

Initializing & Removing Key, Value Pairs

- Initializing From a List

```
%translate = ( 'atg' => 'M',
               'taa' => '*',
               'ctt' => 'L',
               'cct' => 'P' );
```

- Removing key-value pairs

```
delete $translate{'taa'};
```

Checking if a key exists

```
if (exists $translate{‘atg’}) {
   print “Methionine found in translation table\n”;
}
else {
   print “Methionine not found in translation table\n”;
}
if (exists $translate{‘ata’}) {
   print “Isoleucine found in translation table\n”;
}
else {
   print “Isoleucine not found in translation table\n”;
}
```

Methionine found in translation table
Isoleucine not found in translation table

Reaching into a hash

```
my @codons = keys %translate;
print "@codons" , " - all keys\n";
```

atg ctt taa - all keys

```
my @aa = values %translate;
print "@aa" , " - all values\n";
```

M L * - all values

Iterating Through Hash Contents

- First get all the keys from the hash

```
my @keys = keys %translate;
```

- Using a "foreach" loop

```
foreach my $key (@keys) {
    print "The AA code for ", $key, " is ", $translate{$key}, "\n";
}
```

- Using a "for" loop

```
for (my $index = 0; $index < scalar(@keys); ++$index) {
    my $key = $keys[$index];
    print "The AA code for ", $key, " is ", $translate{$key}, "\n";
}
```

Problem Sets

- Problem #1
  - Exercises 1-3, page 54, Learning Perl
- Problem #2
  - Exercises 1, page 105, Learning Perl
  - How the program (call it names.pl) in exercise 1 works:

```
$ names.pl fred
  flinstone
$ names.pl barney
  rubble
$ names.pl wilma
  flinstone
```

References

- Perl docs online perldoc.perl.org
- Learning Perl. Schwartz and Christiansen
  - Chapters 3 & 6
- Programming Perl. Wall, Christiansen and Schwartz
- Effective Perl Programming. Hall and Schwartz

References

Simon Prochnik, Dave Messina, Lincoln Stein, Steve Rozen
PfB 2010

What good are references?

Sometimes you need a more complex data structure than a list.

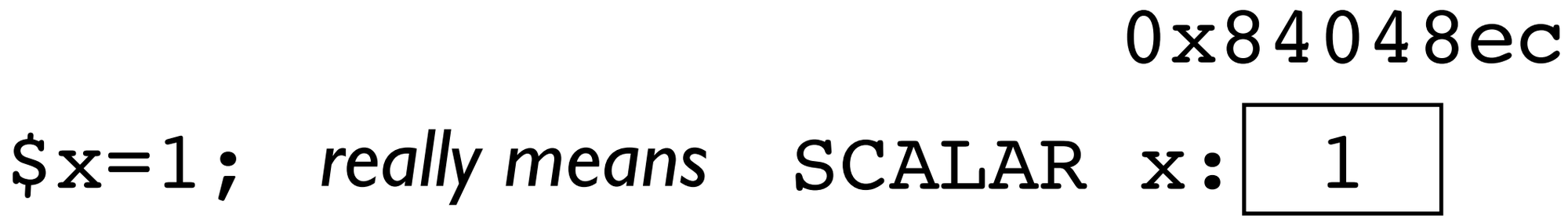
What if you want to keep together several related pieces of information?

Gene	Sequence	Organism
HOXB2	ATCAGCAATATACAATTATAAAGGCCTAAATTTAAAA	mouse
HDAC1	GAGCGGAGCCGCGGGCGGGAGGGCGGACGGAC	human

What is a reference?

Well first, what is a variable?

A variable is a labeled memory address that holds a value. The location's label is the name of the variable.



What is a list?

```
@y = (1, ‘a’, 23);
```

really means

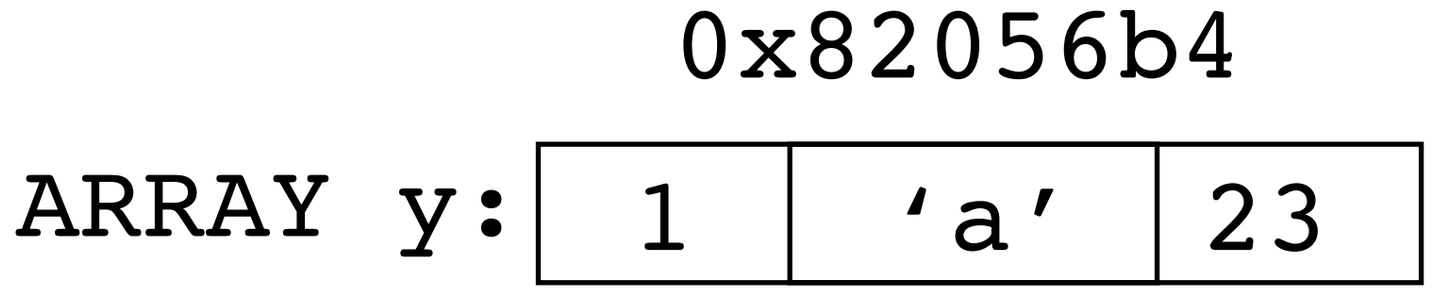
0x82056b4

ARRAY y:

1	‘a’	23

A variable is a labeled memory address.

When we read the contents of the variable, we are reading the contents of the memory address.

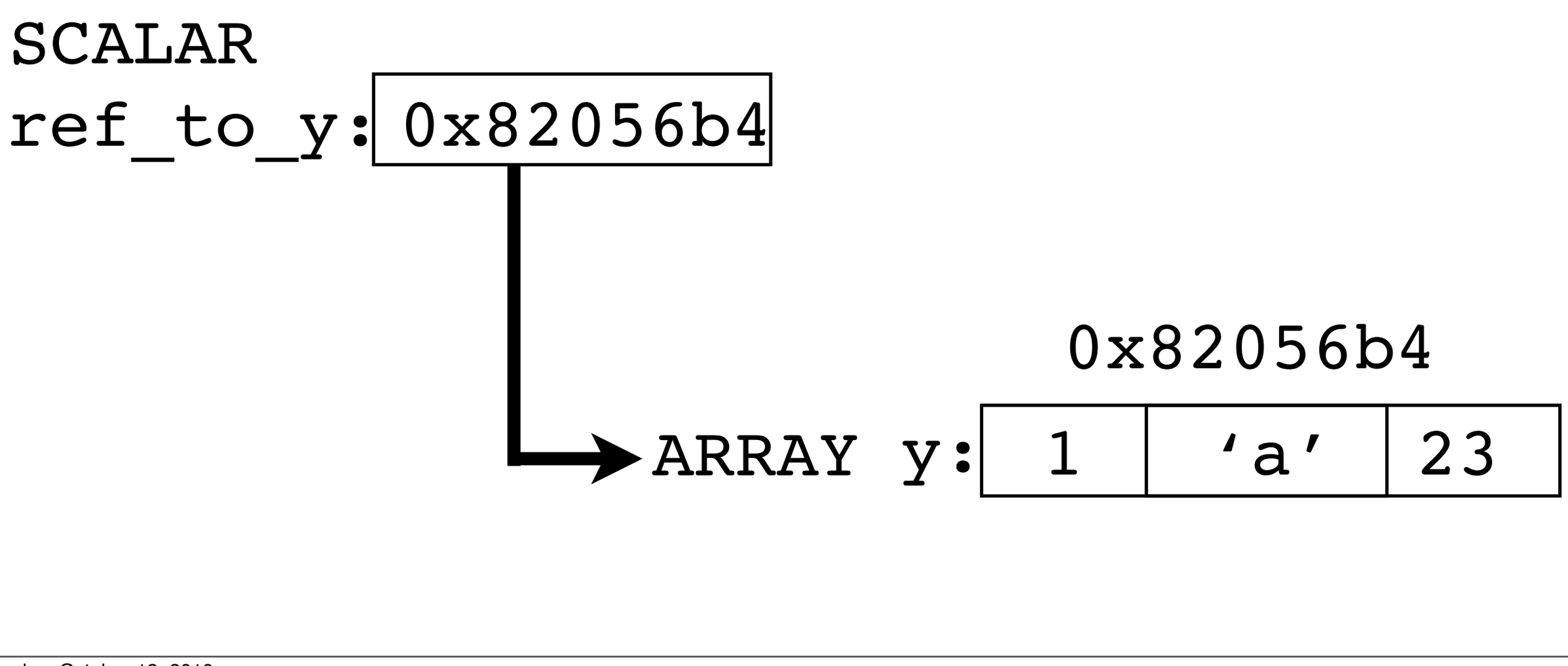


So, what is a reference?

A reference is a variable that contains the memory address of some data.

It does not contain the data itself. It contains the memory address where some data is stored.

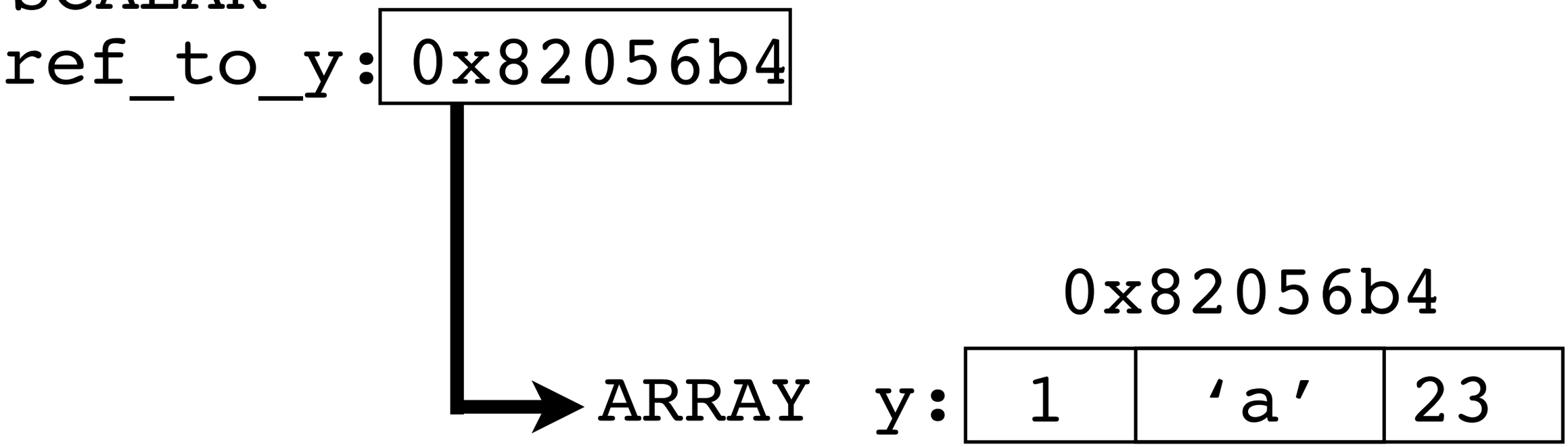
We can create a reference to named variable @y this way:



SCALAR ref_to_y: 0x82056b4

If we try to print out $ref_to_y, we see the raw memory address:

```
print $ref_to_y,"\n";
ARRAY(0x82056b4)
```



To see the contents of what $ref_to_y points to, we have to dereference it:

```
print join ' ',@{$ref_to_y};
1 a 23
```

You can create references to scalars, arrays and hashes

```
# create some references
$scalar_ref = \$count;
$array_ref  = \@array;
$hash_ref   = \%hash;
```

To dereference a reference, place the appropriate symbol ($, @, %) in front of the reference:

```
# dereference your references:
$count_copy  = ${$scalar_ref};
@array_copy  = @{$array_ref};
%hash_copy   = %{$hash_ref};
```

A reference is a pointer to the data. It isn't a copy of the data.

When you make a reference to a variable, you have only created another way to get at the data.

There is still only one copy of the data.

```
@y                    = (1,'a',23);
$ref_to_y             = \@y;
print join ' ',@{$ref_to_y};
```
1 a 23

```
push @{$ref_to_y},'new1','new2';

print join ' ',@y;
```
1 a 23 new1 new2

This is in contrast to doing a direct copy from one variable to another, which creates a new data structure in a new memory location.

```
@y    = (1,'a',23);
@z    = @y;
push @y,'new1','new2';

print join ' ',@y;
```
1 a 23 new1 new2

```
print join ' ',@z;
```
1 a 23

If you have a reference to an array or a hash, you can access any element.

```
$value    = $y[2];
```

directly access the 3rd element in @y

```
$value    = ${$ref_to_y}[2];
```

dereference the reference, then access the 3rd element in @y

```
${$ref_to_y}[2] = 'new';
print join ' ',@y;
```

1 a new

change the value of the 3rd element in @y

```
%z          = (‘dog‘  => 'animal',
               ‘potato’ => 'vegetable',
               ‘quartz’ => 'mineral',
               ‘tomato’ => 'vegetable');

$ref_to_z = \%z;

$value    = $z{‘dog’};
```

directly access the value associated with the key ‘dog’ in the hash %z

```
$value    = ${$ref_to_z}{‘dog’};
```

dereference the reference, then get the value associated with the key ‘dog’ in the hash %z

```
${$ref_to_z}{‘tomato’} = 'fruit';
print $z{‘tomato’}, “\n”;
```
fruit

change the value associated with the key ‘tomato’ in the hash %z

Anonymous Hashes and Arrays

You will not usually make references to existing variables. Instead you will create anonymous hashes and arrays. These have a memory location, but no symbol or name, i.e. you can't write @my_data. The reference is the only way to address them.

To create an anonymous array use the form:

```
$ref_to_arry = ['item1','item2',...]
```

To create an anonymous hash, use the form:

```
$ref_to_hash = {‘key1’=>'val1',‘key2’=>'value2'}
```

```
$y_gene_families = ['DAZ', 'TSPY', 'RBMY', 'CDY1',
'CDY2' ];

$y_gene_family_counts = { 'DAZ'  => 4,
                          'TSPY' => 20,
                          'RBMY' => 10,
                          'CDY2' => 2 };

$third_item_of_array = $y_gene_families->[2];
$daz_count           = $y_gene_family_counts->{DAZ};
```

$y_gene_families is a reference to an array, and $y_gene_family_counts is a reference to a hash.

Making a Hash of Hashes

The beauty of anonymous arrays and hashes is that you can nest them:

```
my %y_gene_data = (
    ‘DAZ’  => {‘family_size’ => 4,
               ‘description’ => 'deleted in azoospermia' },
    ‘TSPY’ => {‘family_size’ => 20,
               ‘description’ => 'testis specific protein'},
    ‘RBMY’ => {‘family_size’ => 10,
               ‘description’ => 'RNA-binding motif Y'},
    ‘CDY2’ => {‘family_size’ => 2,
               ‘description’ => 'chromodomain protein' }
);

# what is the size of the RBMY family?
my $size = $y_gene_data{‘RBMY’}{‘family_size’};

# what is the description of TSPY?
my $desc = $y_gene_data{‘TSPY’}{‘description’};
```

Making an Array of Arrays

```
my @spotarray = (
                 [0.124, 43.2, 0.102, 80.4],
                 [0.113, 60.7, 0.091, 22.6],
                 [0.084, 112.2, 0.144, 35.3]
                );
my $cell_1_0  = $spotarray[1][0];
print $cell_1_0;
```

```
0.113
```

Examining References

Inside a Perl script, the ref function tells you what kind of value a reference points to:

```
print ref($y_gene_data), "\n";
HASH

print ref($spotarray), "\n";
ARRAY

$x = 1;
print ref($x), "\n";
(empty string)
```

Examining complex data structures in the debugger

Inside the Perl debugger, the "x" command will pretty-print the contents of a complex reference:

```
DB<3> x $y_gene_data
0  HASH(0x8404bb0)
   'CDY2' => HASH(0x8404b80)
      'description' => 'chromodomain protein, Y-linked'
      'family_size' => 2
   'DAZ' => HASH(0x84047fc)
      'description' => 'deleted in azoospermia'
      'family_size' => 4
   'RBMY' => HASH(0x8404b50)
      'description' => 'RNA-binding motif Y'
      'family_size' => 10
   'TSPY' => HASH(0x8404b20)
      'description' => 'testis specific protein Y-linked'
      'family_size' => 20
```

Scripting Example: Creating a Hash of Hashes

We are presented with a table of sequences in the following format: the ID of the sequence, followed by a tab, followed by the sequence itself.

```
2L52.1      atgtcaatggtaagaaatgtatcaaatcagagcgaaaaattggaagtaag...
4R79.2      tcaaatacagcaccagctccttttttatagttcgaattaatgtccaact...
AC3.1       atggctcaaactttactatcacgtcatttccgtggtgtcaactgttattt...
...
```

For each sequence calculate the length of the sequence and the count for each nucleotide. Store the results into a hash of hashes in which the outer hash's key is the ID of the sequence, and the inner hashes' keys are the nucleotides and the values are the counts.

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

use strict;

# tabulate nucleotide counts, store into %sequences

my %seqs; # initialize hash
while (my $line = <>) {
  chomp $line;
  my ($id,$sequence) = split "\t",$line;
  my @nucleotides    = split '', $sequence;  # array of base pairs
  foreach my $n (@nucleotides) {
     $seqs{$id}{$n}++; # count nucleotides and keep tally
  }
}

# print table of results
print join("\t",'id','a','c','g','t'),"\n";

foreach my $id (sort keys %seqs) {
   print join("\t",$id,
                   $seqs{$id}{a},
                   $seqs{$id}{c},
                   $seqs{$id}{g},
                   $seqs{$id}{t},
              ),"\n";
}
```

The output will look something like this:

```
id          a    c     g    t
2L52.1     23   4     12    11
4R79.2     15   12    5     18
AC3.1      11   11    8     20
...
```

Perl References

Simon Prochnik, Lincoln Stein (From Steve Rozen, 2001)

Problem Set

1. What kind of data structure could you use to represent the data in the table below?

CDC2	45	liver
PLK1	34.2	heart
MCM4	9	kidney

2. Write a script to generate a data structure which represents the table above.

3. The table below is the same as the table above, but has labels added as headings.

   Modify your script such that the data is now stored according to the labels, so that you can access the data using those labels. For example, if your data structure is called %hash, you should be able to look up the data related the CDC2 gene like this:

```
$gene = 'CDC2';
my $expression_for_gene = $hash{$gene}{'expression'};
my $tissue_for_gene     = $hash{$gene}{'tissue'};
```

gene	expression	tissue
CDC2	45	liver
PLK1	34.2	heart
MCM4	9	kidney

4. Modify your script so that the data for MCM4 is printed out like this:

```
gene      expression     tissue
MCM4      9              kidney
```

Perl6 - Subroutines and Modules

Lincoln Stein

Suggested Reading

Chapters 4 and 11 of *Learning Perl*, especially the section *Using Simple Modules*. Chapter 6 of *Beginning Perl for Bioinformatics*.

Lecture Notes

Subroutines

Modules

Problem Set

1. Write a subroutine to concatenate two strings of DNA.
2. Write a subroutine to report the percentage of each nucleotide in DNA. You've seen the plus operator +. You will also want to use the divide operator / and the multiply operator *. Count the number of each nucleotide, divide by the total length of the DNA, then multiply by 100 to get the percentage. Your arguments should be the DNA and the nucleotide you want to report on. The `int` function can be used to discard digits after the decimal point, if needed.
3. Using the CPAN web site, locate a module for verifying credit card numbers. Download and build it (don't try to install it, because you need root privileges to do this).
4. Using the standard object-oriented Math::BigInt library, which allows you to store really big integers, write a script that will read in a 100+ digit integer and calculate its square root.
5. (extra credit) Create a module that counts the number of times a restriction site appears in a nucleotide string. The exported function should be named count_sites() and should be called like this:

```
$count = count_sites('name_of_site',$nucleotide_string);

# for example
$count = count_sites('ecoRI','GGGATTTGACCGGAATTCCGATCCCAAGGTTC');
```

Hints: Use a parenthesized regular expression and assign the results of a string match to an array. Store the relationships between the name of a restriction site and its regular expression in a hash.

Subroutines

Subroutines are blocks of code that you can call in different places and contexts. Subroutines can take arguments, and return results.

Why is this useful? Because it lets you solve a problem once and then reuse your solution over and over again. For example, say you've written a chunk of code that normalizes a DNA sequence by removing unwanted characters. By turning it into a named subroutine, you can reuse this piece of code over and over again within the same program without cutting and pasting. Later, you'll be able to put this subroutine into a personal code library and reuse it among many scripts.

Subroutines also make scripts shorter and easier to understand.

Example: Cleansing a Sequence

Sequences that come out of GenBank are "contaminated" with line numbers and whitespace, like this:

```
   1 aagacacgga agtagctccg aacaggaaga ggacgaaaaa aataaccgtc cgcgacgccg
  61 agacaaaccg gacccgcaac caccatgaac agcaaaggcc aatatccaac acagccaacc
 121 taccctgtgc agcctcctgg gaatccagta taccctcaga ccttgcatct tcctcaggct
 181 ccaccctata ccgatgctcc acctgcctac tcagagctct atcgtccgag ctttgtgcac
 241 ccaggggctg ccacagtccc caccatgtca gccgcatttc ctggagcctc tctgtatctt
 301 cccatggccc agtctgtggc tgttgggcct ttaggttcca caatccccat ggcttattat
 361 ccagtcggtc ccatctatcc acctggctcc acagtgctgg tggaaggagg gtatgatgca
 421 ggtgccagat ttggagctgg ggctactgct ggcaacattc ctcctccacc tcctggatgc
 481 cctcccaatg ctgctcagct tgcagtcatg cagggagcca acgtcctcgt aactcagcgg
 541 aaggggaact tcttcatggg tggttcagat ggtggctaca ccatctggtg aggaaccaag
 601 gccacctttg tgccgggaaa gacatcacat accttcagca cttctcacaa tgtaactgct
 661 ttagtcatat taacctgaag ttgcagttta gacacatgtt gttggggtgt ctttctggtg
 721 cccaaacttt caggcacttt tcaaatttaa taaggaacca tgtaatggta gcagtacctc
 781 cctaaagcat tttgaggtag gggaggtatc cattcataaa atgaatgtgg gtgaagccgc
 841 cctaaggatt ttcctttaat ttctctggag taatactgta ccatactggt ctttgctttt
 901 agtaataaaa catcaaatta ggtttggagg gaactttgat cttcctaaga attaaagttg
 961 ccaaattatt ctgattggtc tttaatctcc tttaagtctt tgatatatat tactttataa
1021 atggaacgca ttagttgtct gccttttcct ttccatccct tgccccaccc atcccatctc
1081 caaccctagt c
```

You want to remove all this extraneous stuff and turn the sequence into a single long string:

```
aagacacggaagtagctccgaacaggaagaggacgaaaaaaataaccgtccgcgacgccgagacaaaccggacccgc
```

To do this, you've written several statements that lowercase the sequence, and remove whitespace. If the sequence contains unexpected characters after this, we die:

```
$sequence = lc $sequence;  # translate everything into lower case
$sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
$sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid characters!";
```

We can turn this into a named subroutine with the following three steps:

1. Turn it into a block:

```
{
  $sequence = lc $sequence;  # translate everything into lower case
  $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
  $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid cha
}
```

2. Label the block with `sub` *`subroutine_name`*:

```
sub cleanup_sequence
```

```
{
  $sequence = lc $sequence;  # translate everything into lower case
  $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
  $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid char
}
```

3. Add statements to read the subroutine argument(s) and return the subroutine result(s):

```
sub cleanup_sequence
{
  my ($sequence) = @_;
  $sequence = lc $sequence;  # translate everything into lower case
  $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
  $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid char
  return $sequence;
}
```

`cleanup_sequence()` now acts like a built-in function. It takes a list of arguments (in this case only one, the original sequence) and returns a list of results (in this case, only one, the cleaned-up sequence):

```
my $seq;
while (my $seqline = <IN>) { # read sequence from a file
  my $clean   = cleanup_sequence($seqline);  # clean it up
  $seq       .= $clean;                      # add it to full sequence
}
```

Getting Data in and out of a Subroutine

When you invoke a subroutine, you pass it a list of arguments and receive a list of results:

```
my @results = my_subroutine('arg1','arg2','arg3'...);
```

You'll now see how subroutines can retrieve its arguments and return its results.

Getting the Subroutine Arguments

Within the subroutine, the arguments are passed to it in an automatic ("magic") array variable named `@_`. One common idiom is for the first statement in a subroutine to copy `@_` into a list of named variables:

```
sub my_subroutine {
   my ($arg1,$arg2,$arg3) = @_;
   ...
}
```

Returning the Subroutine Results

To return a list of results from a subroutine to its caller, use the **return** operator. Usually you will call **return** at the very end of the subroutine, but you can call it earlier in special cases if you want to exit the subroutine earlier.

This subroutine will add a PCR primer sequence to the beginning and end of a DNA sequence and return the result.

```
sub add_linkers {
   my ($linker,$sequence) = @_;
```

```
    my $reverse_linker = $linker;
    $reverse_linker    =~ tr/gatcGATC/ctagCTAG/;  # reverse complement it
    my $result         = $linker . $sequence . $reverse_linker;
    return $result;
}
```

You can return a single value, a list, or nothing:

```
    return $single_value;  # scalar
    return ('a','long','list','of','items');  # list
    return @an_array;      # list contained in an array variable
    return;                # return empty list or undef, depending on context
```

Subroutine Anatomy

Anatomy of a Subroutine

Lastly, the age old question, *Where do you put the subroutines in your script?*. Usually the subroutine definitions go at the bottom of the script, following the last statement.

To visually separate the statements from the subroutine, you can add a comment line if you like.

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

# comments describing what the script does
# more comments, including author and script name

my ($variables, $variables, @more_variables);   # declare some variables

while (my $line = <IN>) {
   my @results = subroutine_1();
   my $result  = subroutine_2(\@results);
}

do_something_at_end_of_script;


### Subroutines ###

sub subroutine_1 {
   my ($local1,$local2,$local3) = @_;
   do_something;
}

sub subroutine_2 {
   my ($local1,$local2,$local3) = @_;
   do_something;
}
```

Modules

Using a Module

A module is a package of useful subroutines and variables that someone has put together. Modules extend the ability of Perl.

Example 1: The File::Basename Module

The **File::Basename** module is a standard module that is distributed with Perl. When you load the **File::Basename** module, you get two new functions, *basename* and *dirname*.

basename takes a long UNIX path name and returns the file name at the end. *dirname* takes a long UNIX path name and returns the directory part.

```
#!/usr/bin/perl
# file: basename.pl

use strict;
use File::Basename;

my $path = '/bush_home/bush1/lstein/C1829.fa';
my $base = basename($path);
my $dir  = dirname($path);

print "The base is $base and the directory is $dir.\n";
```

The output of this program is:

```
The base is C1829.fa and the directory is /bush_home/bush1/lstein.
```

The **use** function loads up the module named *File::Basename* and **imports** the two functions. If you didn't use **use**, then the program would print an error:

```
Undefined subroutine &main::basename called at basename.pl line 8.
```

Example 2: The Env Module

The **Env** module is a standard module that provides access to the environment variables. When you load it, it **imports** a set of scalar variables corresponding to your environment.

```
#!/usr/bin/perl
# file env.pl

use strict;
use Env;

print "My home is $HOME\n";
print "My path is $PATH\n";
print "My username is $USER\n";
```

When this runs, the output is:

```
My home is /bush_home/bush1/lstein
```

```
My path is /net/bin:/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin:/bush_home/bu
My username is lstein
```

Finding out What Modules are Installed

Here are some tricks for finding out what modules are installed.

Preinstalled Modules

To find out what modules come with perl, look in Appendix A of *Perl 5 Pocket Reference*. From the command line, use the **perldoc** command from the UNIX shell. All the Perl documentation is available with this command:

```
% perldoc perlmodlib
PERLMODLIB(1)  User Contributed Perl Documentation  PERLMODLIB(1)



NAME
       perlmodlib - constructing new Perl modules and finding
       existing ones

DESCRIPTION
THE PERL MODULE LIBRARY
       Many modules are included the Perl distribution.  These
       are described below, and all end in .pm.  You may discover
...
       Standard Modules

       Standard, bundled modules are all expected to behave in a
       well-defined manner with respect to namespace pollution
       because they use the Exporter module.  See their own docu-
       mentation for details.

       AnyDBM_File Provide framework for multiple DBMs

       AutoLoader  Load subroutines only on demand

       AutoSplit   Split a package for autoloading

       B           The Perl Compiler
...
```

To learn more about a module, run **perldoc** with the module's name:

```
% perldoc File::Basename

NAME
       fileparse - split a pathname into pieces

       basename - extract just the filename from a path

       dirname - extract just the directory from a path

SYNOPSIS
            use File::Basename;
```

```
            ($name,$path,$suffix) = fileparse($fullname,@suffixlist)
            fileparse_set_fstype($os_string);
            $basename = basename($fullname,@suffixlist);
            $dirname = dirname($fullname);
...
```

Optional Modules that You May Have Installed

perldoc perllocal will list the names of locally installed modules.

```
% perldoc perllocal
       Thu Apr 27 16:01:31 2000: "Module" the DBI manpage


       o   "installed into: /usr/lib/perl5/site_perl"

       o   "LINKTYPE: dynamic"

       o   "VERSION: 1.13"

       o   "EXE_FILES: dbish dbiproxy"

       Thu Apr 27 16:01:41 2000: "Module" the Data::ShowTable
       manpage


       o   "installed into: /usr/lib/perl5/site_perl"

       o   "LINKTYPE: dynamic"

       o   "VERSION: 3.3"

       o   "EXE_FILES: showtable"

       Tue May 16 18:26:27 2000: "Module" the Image::Magick man-
       page
...
```

But often it's just easier to test directly using perl itself:

```
% perl -e 'use File::Basename;'
%
```

If you get no error when you try to `use` the module, then the module is installed.

Installing Modules

You can find thousands of Perl Modules on CPAN, the Comprehensive Perl Archive Network:

http://www.cpan.org

Installing Modules Manually

Search for the module on CPAN using the keyword search. When you find it, download the .tar.gz module. Then install it like this:

```
% tar zxvf bioperl-1.6.1.tar.gz
bioperl-1.6.1/
bioperl-1.6.1/Bio/
bioperl-1.6.1/Bio/DB/
bioperl-1.6.1/Bio/DB/Ace.pm
bioperl-1.6.1/Bio/DB/GDB.pm
bioperl-1.6.1/Bio/DB/GenBank.pm
bioperl-1.6.1/Bio/DB/GenPept.pm
bioperl-1.6.1/Bio/DB/NCBIHelper.pm
bioperl-1.6.1/Bio/DB/RandomAccessI.pm
bioperl-1.6.1/Bio/DB/SeqI.pm
bioperl-1.6.1/Bio/DB/SwissProt.pm
bioperl-1.6.1/Bio/DB/UpdateableSeqI.pm
bioperl-1.6.1/Bio/DB/WebDBSeqI.pm
bioperl-1.6.1/Bio/AlignIO.pm

%  perl Makefile.PL
Generated sub tests. go make show_tests to see available subtests
...
Writing Makefile for Bio

% make
cp Bio/Tools/Genscan.pm blib/lib/Bio/Tools/Genscan.pm
cp Bio/Root/Err.pm blib/lib/Bio/Root/Err.pm
cp Bio/Annotation/Reference.pm blib/lib/Bio/Annotation/Reference.pm
cp bioback.pod blib/lib/bioback.pod
cp Bio/AlignIO/fasta.pm blib/lib/Bio/AlignIO/fasta.pm
cp Bio/Location/NarrowestCoordPolicy.pm blib/lib/Bio/Location/NarrowestCoordI
cp Bio/AlignIO/clustalw.pm blib/lib/Bio/AlignIO/clustalw.pm
cp Bio/Tools/Blast/Run/postclient.pl blib/lib/Bio/Tools/Blast/Run/postclient.
cp Bio/LiveSeq/Intron.pm blib/lib/Bio/LiveSeq/Intron.pm
...
Manifying blib/man3/Bio::LiveSeq::Exon.3
Manifying blib/man3/Bio::Location::CoordinatePolicyI.3
Manifying blib/man3/Bio::SeqFeature::Similarity.3

% make test
PERL_DL_NONLAZY=1 /net/bin/perl -Iblib/arch -Iblib/lib
 -I/net/lib/perl5/5.6.1/i686-linux -I/net/lib/perl5/5.6.1 -e 'use
 Test::Harness qw(&runtests $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/AAChange.........ok
t/AAReverseMutate...ok
t/AlignIO..........ok
t/Allele...........ok
...
t/WWW..............ok
All tests successful, 95 subtests skipped.
Files=60, Tests=1011, 35 wallclock secs (25.47 cusr +  1.60 csys = 27.07 CPU)

% make install
Installing /net/lib/perl5/site_perl/5.6.1/bioback.pod
Installing /net/lib/perl5/site_perl/5.6.1/biostart.pod
Installing /net/lib/perl5/site_perl/5.6.1/biodesign.pod
```

```
Installing /net/lib/perl5/site_perl/5.6.1/bptutorial.pl
...
```

Installing Modules Using the CPAN Shell

Perl has a CPAN module installer built into it. You run it like this:

```
% cpan

cpan shell -- CPAN exploration and modules installation (v1.59_54)
ReadLine support enabled

cpan>
```

From this shell, there are commands for searching for modules, downloading them, and installing them.

[The first time you run the CPAN shell, it will ask you a lot of configuration questions. Generally, you can just hit return to accept the defaults. The only trick comes when it asks you to select CPAN mirrors to download from. Choose any ones that are in your general area on the Internet and it will work fine.]

Here is an example of searching for the Text::Wrap program and installing it:

```
cpan> i /Wrap/
Going to read /bush_home/bush1/lstein/.cpan/sources/authors/01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok
Going to read /bush_home/bush1/lstein/.cpan/sources/modules/02packages.detail
  Database was generated on Tue, 16 Oct 2001 22:32:59 GMT
CPAN: HTTP::Date loaded ok
Going to read /bush_home/bush1/lstein/.cpan/sources/modules/03modlist.data.gz
Distribution    B/BI/BINKLEY/CGI-PrintWrapper-0.8.tar.gz
Distribution    C/CH/CHARDIN/MailQuoteWrap0.01.tgz
Distribution    C/CJ/CJM/Text-Wrapper-1.000.tar.gz
...
Module          Text::NWrap     (G/GA/GABOR/Text-Format0.52+NWrap0.11.tar.gz)
Module          Text::Quickwrap (Contact Author Ivan  Panchenko )
Module          Text::Wrap      (M/MU/MUIR/modules/Text-Tabs+Wrap-2001.0929.t
Module          Text::Wrap::Hyphenate (Contact Author Mark-Jason Dominus )
Module          Text::WrapProp  (J/JB/JBRIGGS/Text-WrapProp-0.03.tar.gz)
Module          Text::Wrapper   (C/CJ/CJM/Text-Wrapper-1.000.tar.gz)
Module          XML::XSLT::Wrapper (M/MU/MULL/XML-XSLT-Wrapper-0.32.tar.gz)
41 items found

cpan> install Text::Wrap
Running install for module Text::Wrap
Running make for M/MU/MUIR/modules/Text-Tabs+Wrap-2001.0929.tar.gz
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://archive.progeny.com/CPAN/authors/id/M/MU/MUIR/modules/Text-Tabs+Wrap-
CPAN: MD5 loaded ok
Fetching with LWP:
  ftp://archive.progeny.com/CPAN/authors/id/M/MU/MUIR/modules/CHECKSUMS
Checksum for /bush_home/bush1/lstein/.cpan/sources/authors/id/M/MU/MUIR/modul
Scanning cache /bush_home/bush1/lstein/.cpan/build for sizes
Text-Tabs+Wrap-2001.0929/
Text-Tabs+Wrap-2001.0929/MANIFEST
Text-Tabs+Wrap-2001.0929/CHANGELOG
Text-Tabs+Wrap-2001.0929/Makefile.PL
Text-Tabs+Wrap-2001.0929/t/
```

```
Text-Tabs+Wrap-2001.0929/t/fill.t
Text-Tabs+Wrap-2001.0929/t/tabs.t
Text-Tabs+Wrap-2001.0929/t/wrap.t
Text-Tabs+Wrap-2001.0929/README
Text-Tabs+Wrap-2001.0929/lib/
Text-Tabs+Wrap-2001.0929/lib/Text/
Text-Tabs+Wrap-2001.0929/lib/Text/Wrap.pm
Text-Tabs+Wrap-2001.0929/lib/Text/Tabs.pm

  CPAN.pm: Going to build M/MU/MUIR/modules/Text-Tabs+Wrap-2001.0929.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for Text
cp lib/Text/Wrap.pm blib/lib/Text/Wrap.pm
cp lib/Text/Tabs.pm blib/lib/Text/Tabs.pm
Manifying blib/man3/Text::Wrap.3
Manifying blib/man3/Text::Tabs.3
  /usr/bin/make  -- OK
Running make test
PERL_DL_NONLAZY=1 /net/bin/perl -Iblib/arch -Iblib/lib
-I/net/lib/perl5/5.6.1/i686-linux -I/net/lib/perl5/5.6.1 -e 'use
Test::Harness qw(&runtests $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/fill.............ok
t/tabs.............ok
t/wrap.............ok
All tests successful.
Files=3, Tests=37,  0 wallclock secs ( 0.20 cusr +  0.00 csys =  0.20 CPU)
  /usr/bin/make test -- OK
Running make install
Installing /net/lib/perl5/5.6.1/Text/Wrap.pm
Installing /net/man/man3/Text::Wrap.3
Installing /net/man/man3/Text::Tabs.3
Writing /net/lib/perl5/5.6.1/i686-linux/auto/Text/.packlist
Appending installation info to /net/lib/perl5/5.6.1/i686-linux/perllocal.pod
  /usr/bin/make install UNINST=1 -- OK

cpan> quit
Lockfile removed.
```

Where are Modules Installed?

Module files end with the extension .pm. If the module name is a simple one, like **Env**, then Perl will look for a file named **Env.pm**. If the module name is separated by :: sections, Perl will treat the :: characters like directories. So it will look for the module **File::Basename** in the file **File/Basename.pm**

Perl searches for module files in a set of directories specified by the Perl library path. This is set when Perl is first installed. You can find out what directories Perl will search for modules in by issuing **perl -V** from the command line:

```
% perl -V
Summary of my perl5 (revision 5.0 version 6 subversion 1) configuration:
 Platform:
   osname=linux, osvers=2.4.2-2smp, archname=i686-linux
...
 Compiled at Oct 11 2001 11:08:37
```

```
@INC:
  /usr/lib/perl5/5.6.1/i686-linux
  /usr/lib/perl5/5.6.1
  /usr/lib/perl5/site_perl/5.6.1/i686-linux
  /usr/lib/perl5/site_perl/5.6.1
  /usr/lib/perl5/site_perl
  .
```

You can modify this path to search in other locations by placing the **use lib** command somewhere at the top of your script:

```
#!/usr/bin/perl

use lib '/home/lstein/lib';
use MyModule;
...
```

This tells Perl to look in **/home/lstein/lib** for the module MyModule before it looks in the usual places. Now you can install module files in this directory and Perl will find them.

Sometimes you really need to know where on your system a module is installed. Perldoc to the rescue again -- use the `-l` command-line option:

```
% perldoc -l File::Basename
/System/Library/Perl/5.8.8/File/Basename.pm
```

The Anatomy of a Module File

Here is a very simple module file named "MySequence.pm":

```
package MySequence;
#file: MySequence.pm

use strict;
our $EcoRI = 'ggatcc';

sub reverseq {
  my $sequence = shift;
  $sequence = reverse $sequence;
  $sequence =~ tr/gatcGATC/ctagCTAG/;
  return $sequence;
}

sub seqlen {
  my $sequence = shift;
  $sequence =~ s/[^gatcnGATCN]//g;
  return length $sequence;
}

1;
```

A module begins with the keyword **package** and ends with "1;". **package** gives the module a name, and the

1; is a true value that tells Perl that the module compiled completely without crashing.

The **our** keyword declares a variable to be global to the module. It is similar to **my**, but the variable can be shared with other programs and modules ("my" variables cannot be shared outside the current file, subroutine or block). This will let us use the variable in other programs that depend on this module.

To install this module, just put it in the Perl module path somewhere, or in the current directory.

Using the MySequence.pm Module

Using this module is very simple:

```
#!/usr/bin/perl
#file: sequence.pl

use strict;
use MySequence;

my $sequence = 'gattccggatttccaaagggttcccaatttggg';
my $complement = MySequence::reverseq($sequence);

print "original   = $sequence\n";
print "complement = $complement\n";
```

```
% sequence.pl
original   = gattccggatttccaaagggttcccaatttggg
complement = cccaaattgggaacccctttggaaatccggaatc
```

Unless you explicitly export variables or functions, the calling function must explicitly *qualify* each MySequence function by using the notation:

```
MySequence::function_name
```

For a non-exported variable, the notation looks like this:

```
$MySequence::EcoRI
```

Exporting Variables and Functions from Modules

To make your module export variables and/or functions like a "real" module, use the Exporter module.

```
package MySequence;
#file: MySequence.pm

use strict;
use base 'Exporter';

our @EXPORT    = qw(reverseq seqlen);
our @EXPORT_OK = qw($EcoRI);

our $EcoRI = 'ggatcc';

sub reverseq {
  my $sequence = shift;
  $sequence = reverse $sequence;
```

```
  $sequence =~ tr/gatcGATC/ctagCTAG/;
  return $sequence;
}

sub seqlen {
  my $sequence = shift;
  $sequence =~ s/[^gatcnGATCN]//g;
  return length $sequence;
}

1;
```

The **use base 'Exporter'** line tells Perl that this module is a type of "Exporter" module. As we will see later, this is a way for modules to inherit properties from other modules. The Exporter module (standard in Perl) knows how to export variables and functions.

The **our @EXPORT = qw(reverseq seqlen)** line tells Perl to export the functions **reverseq** and **seqlen** automatically. The **our @EXPORT_OK = qw($EcoRI)** tells Perl that it is OK for the user to import the $EcoRI variable, but not to export it automatically.

The qw() notation is telling Perl to create a list separated by spaces. These lines are equivalent to the slightly uglier:

```
    our @EXPORT    = ('reverseq','seqlen');
```

Using the Better MySequence.pm Module

Now the module exports its **reverseq** and **seqlen** functions automatically:

```
#!/usr/bin/perl
#file: sequence2.pl

use strict;
use MySequence;

my $sequence = 'gattccggatttccaaagggttcccaatttggg';
my $complement = reverseq($sequence);

print "original   = $sequence\n";
print "complement = $complement\n";
```

The calling program can also get at the value of the $EcoRI variable, but he has to ask for it explicitly:

```
#!/usr/bin/perl
#file: sequence3.pl

use strict;
use MySequence;

my $sequence = 'gattccggatttccaaagggttcccaatttggg';
my $complement = reverseq($sequence);

print "original   = $sequence\n";
print "complement = $complement\n";

if ($complement =~ /$EcoRI/) {
```

```
  print "Contains an EcoRI site.\n";
} else {
  print "Doesn't contain an EcoRI site.\n";
}
```

POD - Documenting your code

We've used the # sign to comment out lines in our programs, and they're a good form of documenting our code. But comment are only visible when we look at the code, not when we run the program.

But there are other ways to document your code, and the one most commonly used in Perl is POD. When you add POD to your code, then you can produce your very own `man` page for your program.

Object Oriented Perl or 'OOP'

Simon Prochnik
CSHL 2009

- To understand object-oriented syntax in perl, we need to recap three things: references, subroutines, packages.
- These three elements of perl are recycled with slightly different uses to provide object-oriented programming
- The OOP paradigm provides i) a solid framework for sharing code -- reuse
- and ii) a guarantee or contract or specification for how the code will work and how it can be used -- an interface
- and iii) hides the details of implementation so you only have to know how to use the code, not how it works -- saves you time, quick to learn
- Here we are briefly introducing you to OOP and objects so that you can quickly add code that's already written into your scripts, rather than spend hours re-inventing wheels. Many more people use objects than write them.

Why objects? A programming paradigm

- Objects store data and use methods to do things with that data
- they keep the data separate from the rest of the program to stop people (and/or poorly-written code) from messing with the data. The kind of data you can store in an object is specific to a certain class of object.
- the methods (functionality) are also specific to an object and come with it for free (i.e. someone else wrote them and you can use them)
- Objects look after namespace
  - 'use strict' and 'my' avoid conflicts between two variables with the same name
  - objects avoid conflicts between two subroutines (methods) with the same name
- they are a very convenient way to share code that will actually work in the way you expect

I: Recap references

example of syntax

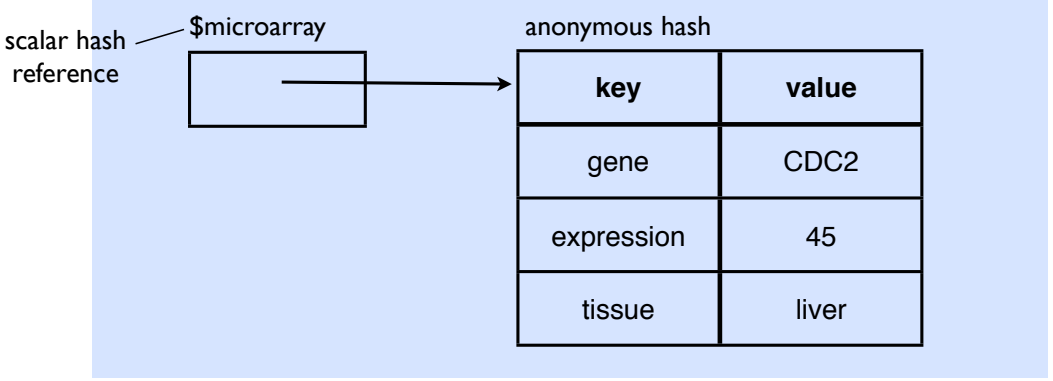
```
$ref_to_hash = {key1=>'value1',key2=>'value2',...}
```

code example

```
my $microarray = {gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
                 };
```

We can store any pieces of data we would like to keep together in a hash

Here is the data structure in memory



II: recap subroutines: example from yesterday

- solve a problem, write code once, and re-use the code
- reusing a single piece of code instead of copying, pasting and modifying reduces the chance you'll make an error and simplifies bug fixing.

```
#!/usr/bin/perl -w
use strict;
my $seq;
while (my $seqline = <>) { # read sequence from standard in
  my $clean   = cleanup_sequence($seqline);  # clean it up
  $seq       .= $clean;                      # add it to full sequence
}
sub cleanup_sequence {
        my ($sequence) = @_;  # set $sequence to first argument
        $sequence = lc $sequence;  # translate everything into lower case
        $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
        $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid
                                    characters!";
        return $sequence;
}
```

III: now let's recap packages

- organise code that goes together into reusable modules, packages

read_clean_sequence.pl

```
#!/usr/bin/perl -w
#File: read_clean_sequence.pl
use strict;
use Sequence;
my $seq;
while (my $seqline = <>) { # read sequence from standard in
  my $clean   = cleanup_sequence($seqline);  # clean it up
  $seq       .= $clean;                      # add it to full sequence
}
```

Sequence.pm

```
#file: Sequence.pm
package Sequence;
use strict;
use base Exporter;
our @EXPORT = ('cleanup_sequence');
sub cleanup_sequence {
        my ($sequence) = @_;  # set $sequence to first argument
        $sequence = lc $sequence;  # translate everything into lower case
        $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
        $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid
characters!";
        return $sequence;
}
1;
```

Let's recap subroutines: new example with references

```
#!/usr/bin/perl -w
use strict;

my $microarray = { gene => 'CDC2,
                   expression => 45,
                   tissue => 'liver',
                 };
...
my $gene_name = gene($microarray);
...
sub gene {
   my ($ref) = @_;
   return $ref->{gene};
}
sub tissue {
   my ($ref) = @_;
   return $ref->{tissue};
}
```

recap packages

main script file

perl module file

script.pl

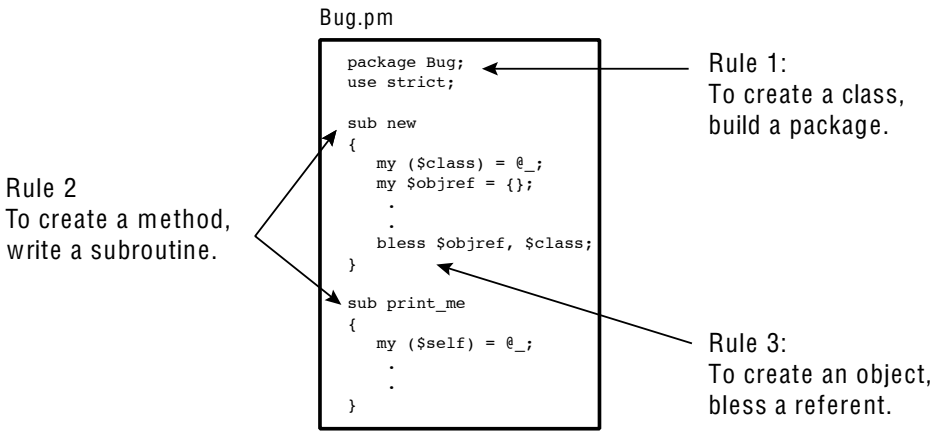
```
#!/usr/bin/perl -w
#File: script.pl
use strict;
use Microarray;

my $microarray = {gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
}
my $gene_name = gene($microarray);
print "Gene for this microarray is
$gene\n";
```

Microaray.pm

```
#File: Microarray.pm
package Microarray;
use strict;
use base 'Exporter';

our @EXPORT = ('gene', 'tissue');

sub gene {
   my ($ref) = @_;
   return $ref->{gene};
}
sub tissue {
   my ($ref) = @_;
   return $ref ->{tissue};
}
1;
```

Three Little Rules

- Rule 1: To create a class, build a package
- Rule 2: To create a method, write a subroutine
- Rule 3: To create an object, bless a reference



Rule 1: To create a class, build a package

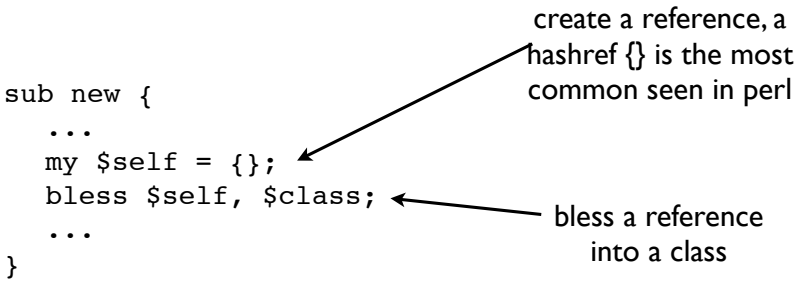
- all the code that goes with an object (methods, special vaiables) goes inside a special package
  - perl packages are just files whose names end with '.pm' e.g. Microarray.pm
  - package filenames should start with a capital letter
  - the name of the perl package tells us the class of the object. This is really the type or kind of object we are dealing with.
- Micorarray.pm is a package, so it will be easy to convert into object-oriented code

Rule 2: To create a method, write a subroutine

- we already have `gene()` in Microarray.pm
- this can be turned into a method
- we need one extra subroutine to create new objects
- the creator method is called `new()` and has one piece of magic...

Rule 3: To create an object, bless a reference

- The `new()` subroutine uses the bless function to create an object
- full details coming up... but here's the skeleton of a `new()` method



recap packages

```
#!/usr/bin/perl -w
#File: script.pl
use strict;
use Microarray;

my $microarray = { gene => 'CDC2',
                   expression => 45,
                   tissue => 'liver',
                 };
my $gene_name = gene($microarray);
print "Gene for this microarray is $gene\n";
```

```
#File: Microarray.pm
package Microarray;
use strict;
use base Exporter;

our @EXPORT = ('gene', 'tissue');

sub gene {
   my $ref = shift;
   return $ref->{gene};
}
sub tissue {
   my $ref = shift;
   return $ref ->{tissue};
}
1;
```

transforming a package into an object-oriented module or class

procedural perl package
(what you saw yesterday)

```
#File: Microarray.pm
package Microarray;
use strict;
use base Exporter;

our @EXPORT = ('gene', 'tissue');

sub gene {
   my ($ref) = @_;
   return $ref->{gene};
}
sub tissue {
   my ($ref) = @_;
   return $ref ->{tissue};
}
1;
```

...transforming the package into a class...

```
#File: Microarray.pm
package Microarray;
use strict;



sub gene {
   my $self = shift; # same as my ($self) = @_;
   return $self->{gene};
}
sub tissue {
   my $self = shift;
   return $self ->{tissue};
}
1;
```

the new() method

```
sub new {
      my $class = shift;
      my %args = @_;
      my $self = {};
      foreach my $key (keys %args) {
          $self -> {$key} =
                 $args{$key};
      }
      # the quasi-religious magic
      # happens here
      bless $self, $class;
      return $self;
}
```

the first argument is always the class of the object you are making. **perl gives you this as the first argument automatically**

a hash reference is the data structure you build an object from in perl

here we initialise variables in the object (in case there are any)

bless makes the object $self (which is a hash reference) become a member of the class $class

`bless` associates an object with its class

Make an anonymous hash in the debugger

```
$a = {};
p ref $a;
HASH
```

Make a MySequence object in the debugger

```
$self = {};
$class = 'MySequence';
bless $self , $class;

x $self
0  MySequence=HASH(0x18bd7cc)
     empty hash
p ref $a
MySequence
```

final step

object-oriented module or class

```
#File: Microarray.pm
package Microarray;
use strict;

sub new {
   my $class = shift;
   my %args = @_;
   my $self = {};
   foreach my $key (keys %args) {
       $self -> {$key} = $args{$key};
   }
   # the magic happens here
   bless $self, $class;
   return $self;
}

sub gene {
   my $self = shift;
   return $self->{gene};
}
sub tissue {
   my $self = shift;
   return $self ->{tissue};
}
1;
```

OOP script

procedural version

```
#File: script.pl
my $microarray = ( gene => 'CDC2',
                   expression => 45,
                   tissue => 'liver',
                  );
my $gene_name = gene($microarray);
print "Gene for this microarray is $gene\n";
```

OO version

```
#File: OO_script.pl
my $microarray = Microarray->new( gene => 'CDC2',
                                  expression => 45,
                                  tissue => 'liver',
                                  );
my $gene_name = $microarray->gene();
print "Gene for this microarray is $gene_name\n";
my $tissue = $microarray->tissue();
print "The tissue is $tissue\n";
```

Did I mention "code lazy"?

- This lecture has introduced you to object-oriented programming
- We are biologists, not computer scientists.
  - you only really need to understand how to **use** objects
  - use other people's packages (classes) to do exciting things so you don't have to write them yourself see bioperl with > 1,000 modules
  - this saves a lot of time
- create your own modules and objects if you have to
  - check CPAN to see if see if someone has already done it for you

Problems

1) Add a method to Microarray.pm called expression() which returns the expression value

2) Curently calling $a = $m->gene() gets the value of gene in the object $m. Modify the gene() method so that it can also set the value of gene if you call gene () with an argument, e.g.

$m->gene('FOXP1'); # this should set the gene name to 'FOXP1'

print $m->gene(); # this should print the value 'FOXP1'

Further reading

- An object is nothing but a way of tucking away complex behaviours into a neat little easy-to-use bundle. (This is what professors call abstraction.) Smart people who have nothing to do but sit around for weeks on end figuring out really hard problems make these nifty objects that even regular people can use. (This is what professors call software reuse.) Users (well, programmers) can play with this little bundle all they want, but they aren't to open it up and mess with the insides. Just like an expensive piece of hardware, the contract says that you void the warranty if you muck with the cover. So don't do that.

- Just what is an object *really*; that is, what's its fundamental type? The answer to the first question is easy. An object is different from any other data type in Perl in one and only one way: you may dereference it using not merely string or numeric subscripts as with simple arrays and hashes, but with named subroutine calls. In a word, with *methods*.

- The answer to the second question is that it's a reference, and not just any reference, mind you, but one whose referent has been *bless*()ed into a particular class (read: package). What kind of reference? Well, the answer to that one is a bit less concrete. That's because in Perl the designer of the class can employ any sort of reference they'd like as the underlying intrinsic data type. It could be a scalar, an array, or a hash reference. It could even be a code reference. But because of its inherent flexibility, an object is usually a hash reference.

- taken from http://perl.about.com

Bioperl I

Sofia Robb
University of Utah

What is Bioperl?

Collection of tools to help you get your work done

Open source, contributed by users

Used by GMOD, wormbase, flybase, me, you

http://www.bioperl.org

Why use BioPerl?

Code is already written.
Manipulate sequences.
Run programs (e.g., blast, clustalw and phylip).
Parsing program output (e.g., blast and alignments).
And much, much more. (http://www.bioperl.org/wiki/Bptutorial.pl)

Learning about bioperl:

- Navigating bioperl website
- Deobfuscator
- Bioperl docs

Manipulation of sequences from a file

Query a local fasta file

Query a remote database

Creating a sequence record

File format conversions

Retrieving annotations

Learning about Bioperl:

Navigating Bioperl website
Deobfuscator
Bioperl docs

www.bioperl.org Main Page

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial

HOWTOs

HOWTOs are narrative-based descriptions of BioPerl modules focusing more on a concept or a task than one specific module.

BioPerl HOWTOs

Beginners HOWTO

An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST, retrieving from databases, and more.

SeqIO HOWTO

Sequence file I/O, with many script examples.

SearchIO HOWTO

Parsing reports from sequence comparison programs like BLAST and writing custom reports.

Tiling HOWTO

Using search reports parsed by SearchIO to obtain robust overall alignment statistics

Feature-Annotation HOWTO

Reading and writing detailed data associated with sequences.

SimpleWebAnalysis HOWTO

Submitting sequence data to Web forms and retrieving results.

Flat Databases HOWTO

Indexing local sequence files for fast retrieval.

PAML HOWTO

Using the PAML package using BioPerl.

OBDA Access HOWTO

BioPerl

howto | discussion | view source | history

HOWTO:Beginners

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator
- Browse Modules

community

- News
- Mailing lists
- Supporting BioPerl

Contents [hide]

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator
- Browse Modules

community

- News
- Mailing lists
- Supporting BioPerl
- BioPerl Media

Deobfuscator

Contents [hide]

What is the Deobfuscator?

The Deobfuscator was written to make it easier to determine the methods that are available from a given BioPerl module (a common BioPerl FAQ).

BioPerl is a highly object-oriented software package, with often multiple levels of inheritance. Although each individual module is usually well-documented for the methods specific to it, identifying the inherited methods is less straightforward.

The Deobfuscator indexes all of the BioPerl POD documentation, taking account of the inheritance tree (thanks to Class::Inspector), and then presents all of the methods available to each module through a searchable web interface.

Where can I find the Deobfuscator?

The Deobfuscator is currently available here, indexing *bioperl-live*.

Welcome to the BioPerl Deobfuscator [bioperl-live] what is it?

Search **class names** by string or Perl regex (examples: Bio::SeqIO, seq, fasta$)

blast Submit Query

OR select a class from the list:

Bio::SearchIO::blast	Event generator for event based parsing of blast reports
Bio::SearchIO::blast_pull	A parser for BLAST output
Bio::SearchIO::blasttable	Driver module for SearchIO for parsing NCBI -m 8/9 format
Bio::SearchIO::blastxml	A SearchIO implementation of NCBI Blast XML parsing.
Bio::SearchIO::megablast	a driver module for Bio::SearchIO to parse megablast reports (format 0)
Bio::Tools::Run::RemoteBlast	Object for remote execution of the NCBI Blast via HTTP
Bio::Tools::Run::StandAloneBlast	Object for the local execution of the NCBI BLAST program suite (blastall, blastpgp, bl2seq). There is experimental support for WU-Blast and NCBI rpsblast.
Bio::Tools::Run::StandAloneNCBIBlast	Object for the local execution of the NCBI BLAST program suite (blastall, blastpgp, bl2seq). With experimental support for NCBI rpsblast.

Deobfuscator

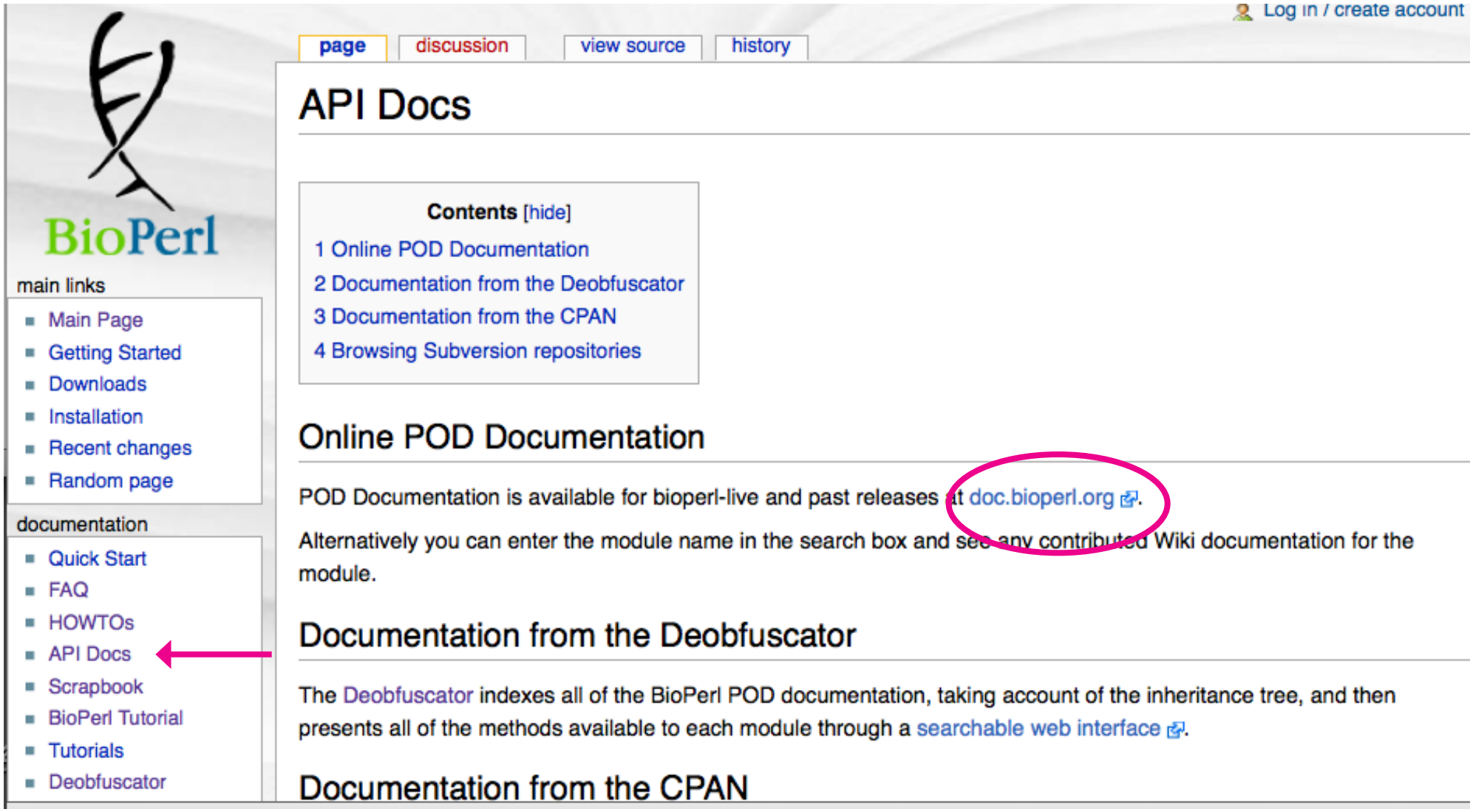
Bio::SearchIO::XML::BlastHandler	XML Handler for NCBI Blast XML parsing.
Bio::SearchIO::XML::PsiBlastHandler	XML Handler for NCBI Blast PSIBLAST XML parsing.

sort by method

methods for **Bio::Tools::Run::StandAloneBlast**

executable	Bio::Tools::Run::StandAloneBlast	string representing the full path to the exe	my $exe = $blastfactory->executable('blasta
finally	Bio::Root::Root	not documented	not documented
io	Bio::Tools::Run::WrapperBase	Bio::Root::IO object	$obj->io($newval)
new	Bio::Tools::Run::StandAloneBlast	Bio::Tools::Run::StandAloneNCBIBlast or StandAloneWUBlast	my $obj = Bio::Tools::Run::StandAloneBlast
no_param_checks	Bio::Tools::Run::WrapperBase	value of no_param_checks	$obj->no_param_checks($newva
otherwise	Bio::Root::Root	not documented	not documented
outfile_name	Bio::Tools::Run::WrapperBase	string	my $outfile = $wrapper->outfile_
program	Bio::Tools::Run::StandAloneBlast	not documented	not documented

doc.bioperl.org



Perldoc (Pdoc rendered) documentation for BioPerl Modules

Released Code

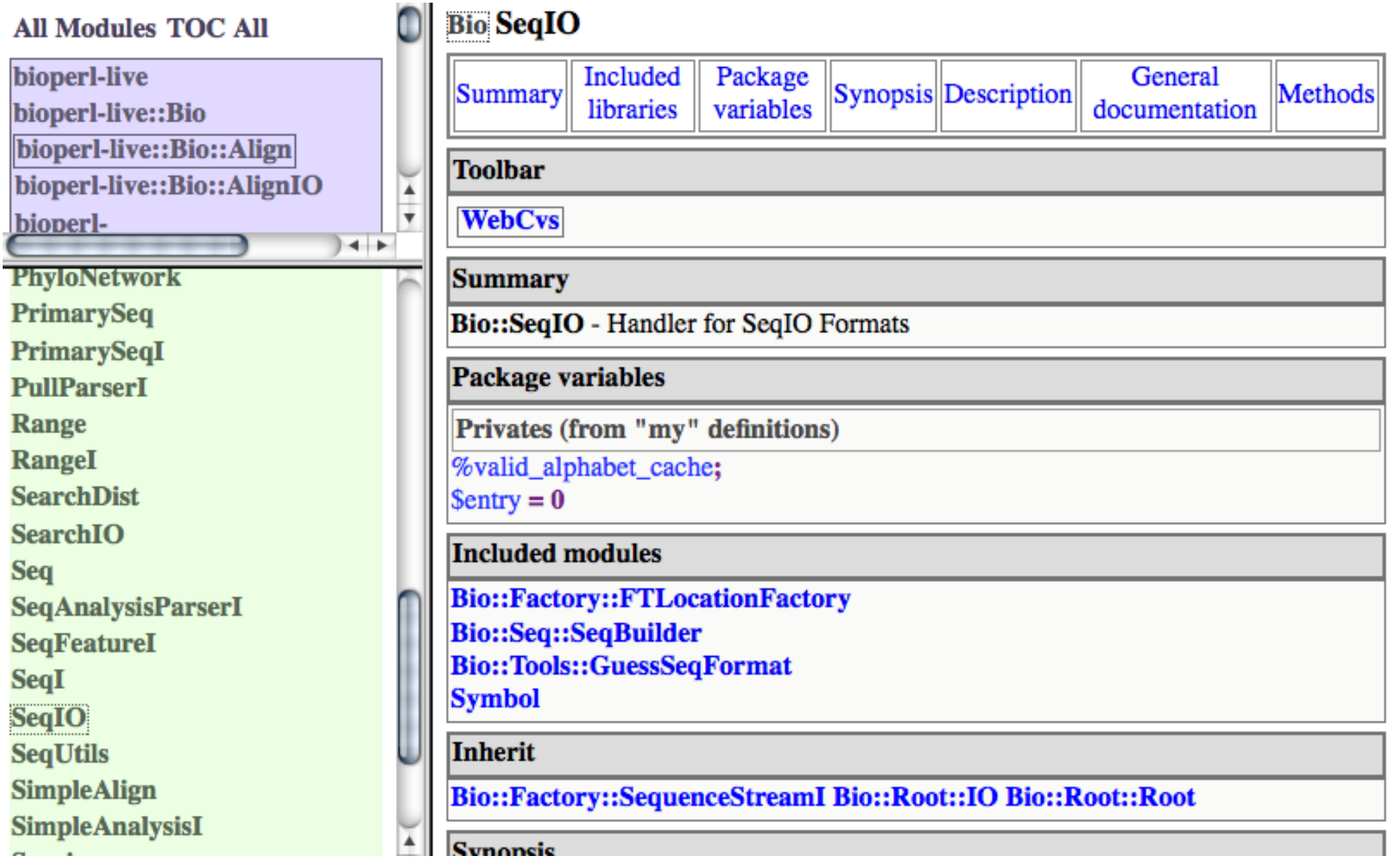
Official documentation for released code is available here:

- BioPerl 1.6.0, download the entire doc set here.
- BioPerl 1.5.2, download the entire doc set here.
- BioPerl 1.5.1, download the entire doc set here.
- BioPerl 1.4, download the entire doc set here.
- BioPerl 1.2.3, download the entire doc set here.
- BioPerl 1.2.2, download the entire doc set here.
- BioPerl 1.2, download the entire doc set here.
- BioPerl 1.0.2, download the entire doc set here.
- BioPerl 1.0.1, download the entire doc set here.
- BioPerl 1.0, download the entire doc set here.

Active Code

This documentation represents the active development code and is autogenerated daily from the SVN repository:

Module	Description
bioperl-live	BioPerl Core Code
bioperl-corba-server	BioPerl BioCORBA Server Toolkit (wraps bioperl objects as BioCORBA objects and runs them in an ORBit ORB)
bioperl-corba-client	Bioperl BioCORBA Client Toolkit (wraps BioCORBA objects as bioperl objects)


All Modules TOC All
bioperl-live
bioperl-live::Bio
bioperl-live::Bio::Align
bioperl-live::Bio::AlignIO
bioperl-
PhyloNetwork
PrimarySeq
PrimarySeqI
PullParserI
Range
RangeI
SearchDist
SearchIO
Seq
SeqAnalysisParserI
SeqFeatureI
SeqI
SeqIO
SeqUtils
SimpleAlign
SimpleAnalysisI
Bio SeqIO
Summary
Included libraries
Package variables
Synopsis
Description
General documentation
Methods
Toolbar
WebCvs
Summary
Bio::SeqIO - Handler for SeqIO Formats
Package variables
Privates (from "my" definitions)
%valid_alphabet_cache;
$entry = 0
Included modules
Bio::Factory::FTLocationFactory
Bio::Seq::SeqBuilder
Bio::Tools::GuessSeqFormat
Symbol
Inherit
Bio::Factory::SequenceStreamI Bio::Root::IO Bio::Root::Root
Synopsis

Bio::SeqIO module synopsis

doc.bioperl.org

Synopsis

```
   use Bio::SeqIO;

   $in  = Bio::SeqIO->new(-file => "inputfilename" ,
                          -format => 'Fasta');
   $out = Bio::SeqIO->new(-file => ">outputfilename" ,
                          -format => 'EMBL');

   while ( my $seq = $in->next_seq() ) {
           $out->write_seq($seq);
   }

# Now, to actually get at the sequence object, use the standard Bio::Seq
# methods (look at Bio::Seq if you don't know what they are)

   use Bio::SeqIO;

   $in  = Bio::SeqIO->new(-file => "inputfilename" ,
                          -format => 'genbank');

   while ( my $seq = $in->next_seq() ) {
      print "Sequence ",$seq->id, " first 10 bases ",
            $seq->subseq(1,10), "\n";
   }

# The SeqIO system does have a filehandle binding. Most people find this
```

Bio::SeqIO module description

doc.bioperl.org

Description

Bio::SeqIO is a handler module for the formats in the SeqIO set (eg, Bio::SeqIO::fasta). It is the officially sanctioned way of getting at the format objects, which most people should use.
The **Bio::SeqIO** system can be thought of like biological file handles. They are attached to filehandles with smart formatting rules (eg, genbank format, or EMBL format, or binary trace file format) and can either read or write sequence objects (Bio::Seq objects, or more correctly, Bio::SeqI implementing objects, of which Bio::Seq is one such object). If you want to know what to do with a Bio::Seq object, read **Bio::Seq**.
The idea is that you request a stream object for a particular format. All the stream objects have a notion of an internal file that is read from or written to. A particular SeqIO object instance is configured for either input or output. A specific example of a stream object is the Bio::SeqIO::fasta object.
Each stream object has functions

```
    $stream->next_seq();
```

and

```
    $stream->write_seq($seq);
```

Bio::SeqIO method list

doc.bioperl.org

Methods		
new	Description	Code
newFh	Description	Code
fh	Description	Code
_initialize	No description	Code
next_seq	Description	Code
write_seq	Description	Code
alphabet	Description	Code
_load_format_module	Description	Code
_concatenate_lines	Description	Code
_filehandle	Description	Code
_guess_format	Description	Code
DESTROY	No description	Code
TIEHANDLE	Description	Code
READLINE	No description	Code

Bio::SeqIO new method description

doc.bioperl.org

Methods description

new | **code** | **next** | **Top**

```
Title   : new
Usage   : $stream = Bio::SeqIO->new(-file => $filename,
                                    -format => 'Format')
Function: Returns a new sequence stream
Returns : A Bio::SeqIO stream initialised with the appropriate format
Args    : Named parameters:
            -file => $filename
            -fh => filehandle to attach to
            -format => format

          Additional arguments may be used to set factories and
          builders involved in the sequence object creation. None of
          these must be provided, they all have reasonable defaults.
            -seqfactory   the Bio::Factory::SequenceFactoryI object
            -locfactory   the Bio::Factory::LocationFactoryI object
            -objbuilder   the Bio::Factory::ObjectBuilderI object
```

See Bio::SeqIO::Handler

Manipulation of sequences from a file

Problem:

You have a sequence file and you want to do something to each sequence.

What do you do first?

HowTo:

http://www.bioperl.org/wiki/HOWTOs

main links
- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation
- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial

HOWTOs

HOWTOs are narrative-based descriptions of BioPerl modules focusing more on a concept or a task than one specific module.

BioPerl HOWTOs

Beginners HOWTO

An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST, retrieving from databases, and more.

SeqIO HOWTO

Sequence file I/O, with many script examples.

SearchIO HOWTO

Parsing reports from sequence comparison programs like BLAST and writing custom reports.

Tiling HOWTO

Using search reports parsed by SearchIO to obtain robust overall alignment statistics

Feature-Annotation HOWTO

Reading and writing detailed data associated with sequences.

SimpleWebAnalysis HOWTO

Submitting sequence data to Web forms and retrieving results.

Flat Databases HOWTO

Indexing local sequence files for fast retrieval.

PAML HOWTO

Using the PAML package using BioPerl.

OBDA Access HOWTO

BioPerl

howto | discussion | view source | history

HOWTO:Beginners

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator
- Browse Modules

community

- News
- Mailing lists
- Supporting BioPerl

Contents [hide]

Retrieving a sequence from a file

One beginner's mistake is to not use Bio::SeqIO when working with sequence files. This is understandable in some respects. You may have read about Perl's `open` function, and Bioperl's way of retrieving sequences may look odd and overly complicated, at first. But don't use `open`! Using open immediately forces you to do the parsing of the sequence file and this can get complicated very quickly. Trust the SeqIO object, it's built to open and parse all the common sequence formats, it can read and write to files, and it's built to operate with all the other Bioperl modules that you will want to use.

Let's read the file we created previously, "sequence.fasta", using SeqIO. The syntax will look familiar:

```
#!/bin/perl -w

use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta" );
```

One difference is immediately apparent: there is no **>** character. Just as with with the `open()` function this means we'll be reading from the "sequence.fasta" file. Let's add the key line, where we actually retrieve the Sequence object from the file using the `next_seq` method:

```
#!/bin/perl -w

use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta" );

$seq_obj = $seqio_obj->next_seq;
```

main links

- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation

- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial

HOWTOs

HOWTOs are narrative-based descriptions of BioPerl modules focusing more on a concept or a task than one specific module.

BioPerl HOWTOs

Beginners HOWTO

An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST, retrieving from databases, and more.

SeqIO HOWTO

Sequence file I/O, with many script examples.

SearchIO HOWTO

Parsing reports from sequence comparison programs like BLAST and writing custom reports.

Tiling HOWTO

Using search reports parsed by SearchIO to obtain robust overall alignment statistics

Feature-Annotation HOWTO

Reading and writing detailed data associated with sequences.

SimpleWebAnalysis HOWTO

Submitting sequence data to Web forms and retrieving results.

Flat Databases HOWTO

Indexing local sequence files for fast retrieval.

PAML HOWTO

Using the PAML package using BioPerl.

OBDA Access HOWTO

HOWTO:SeqIO

This HOWTO will teach you about the Bio::SeqIO system for reading and writing sequences of various formats

Contents [hide]

The basics

This section assumes you've never seen BioPerl before, perhaps you're a biologist trying to get some informatio
something about this hot topic, "bioinformatics". Your first script may want to get some information from a file cor

A piece of advice: always use the module Bio::SeqIO! Here's what the first lines of your script might look like:

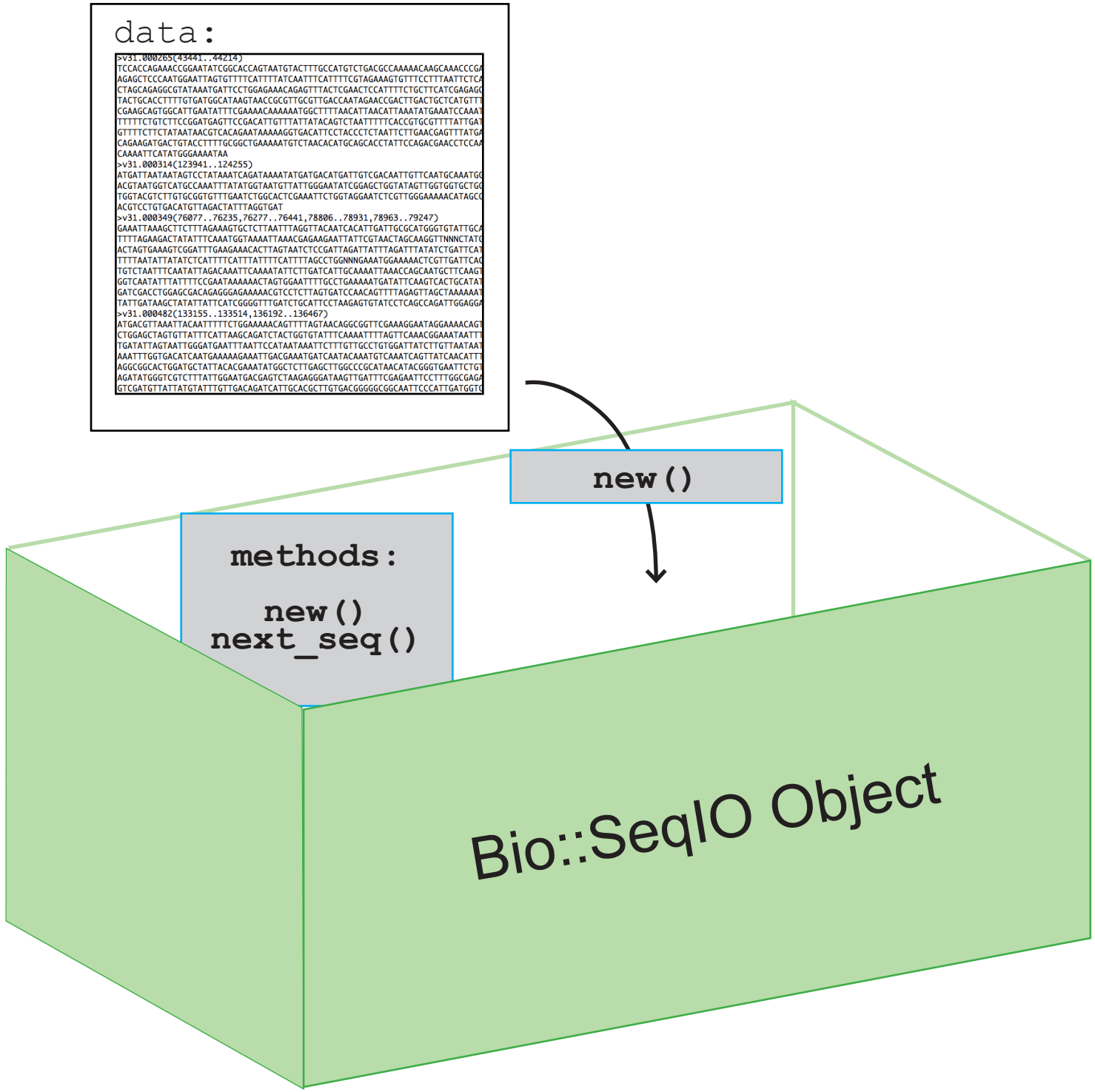
```
#!/bin/perl

use strict;
use Bio::SeqIO;

my $file = shift; # get the file name, somehow
my $seqio_object = Bio::SeqIO->new(-file => $file);
my $seq_object = $seqio_object->next_seq;
```

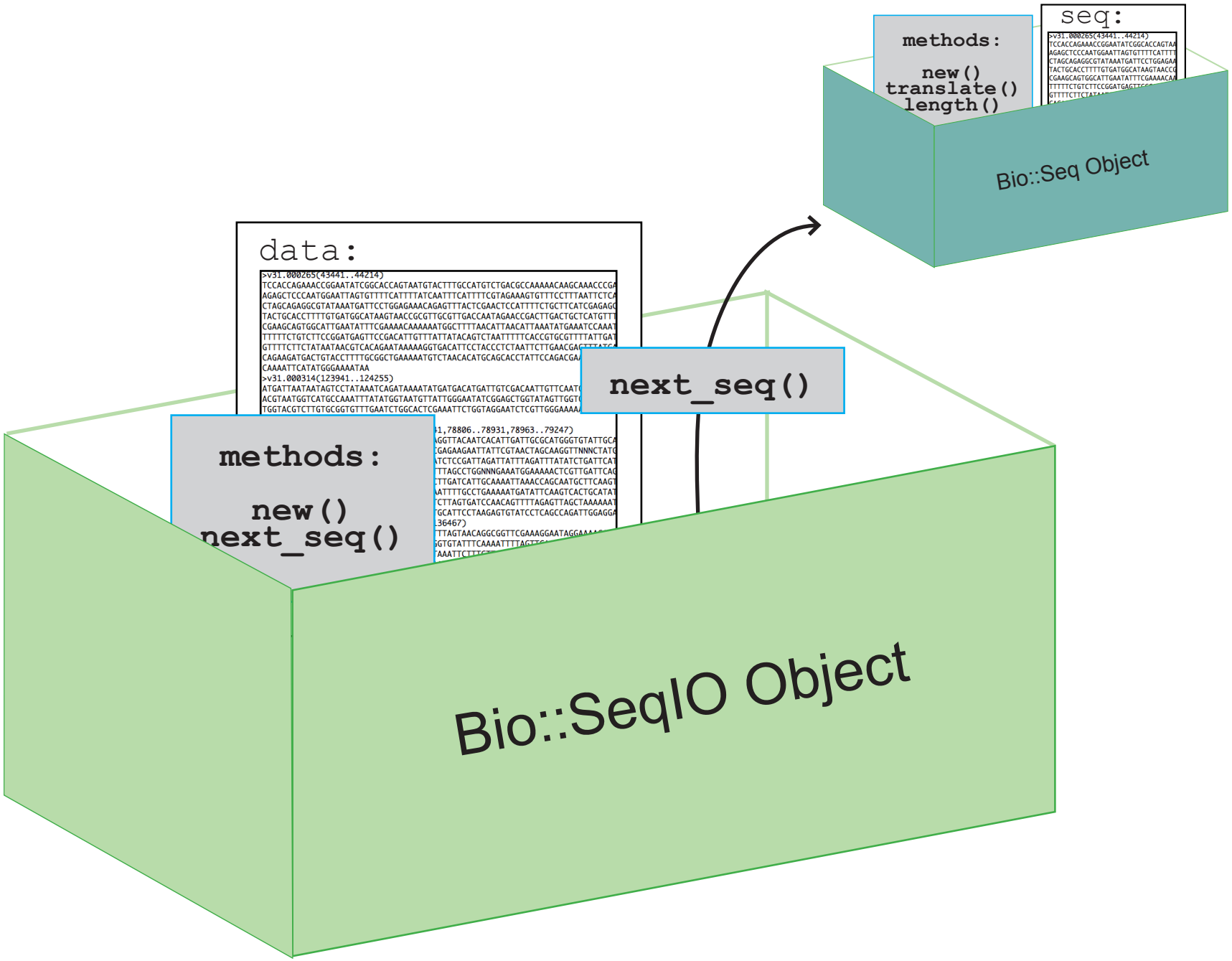
```
#!/usr/bin/perl -w
#file: inFasta_loop.pl
use strict;
use Bio::SeqIO;

my $file = shift;

my $seqIO_object = Bio::SeqIO->new(
                            -file => $file,
                            -format => 'fasta',
                        );

while (my $seq_object = $seqIO_object->next_seq){
        #do stuff to each sequence in the fasta
}
```

What is a SeqIO object?
What is a Seq object?

Objects

Objects are like boxes that hold
your data and
tools (methods) for your data


data:
>v31.000265(43441..44214)
TCCACCAGAAACCGGAATATCGGCACCAGTAATGTACTTTGCCATGTCTGACGCCAAAAACAAGCAAACCCGA
AGAGCTCCCAATGGAATTAGTGTTTTCATTTTATCAATTTCATTTTCGTAGAAAGTGTTTCCTTTAATTCTCA
CTAGCAGAGGCGTATAAATGATTCCTGGAGAAACAGAGTTTACTCGAACTCCATTTTCTGCTTCATCGAGAGC
TACTGCACCTTTTGTGATGGCATAAGTAACCGCGTTGCGTTGACCAATAGAACCGACTTGACTGCTCATGTTT
CGAAGCAGTGGCATTGAATATTTCGAAAACAAAAAATGGCTTTTAACATTAACATTAAATATGAAATCCAAAT
TTTTTCTGTCTTCCGGATGAGTTCCGACATTGTTTATTATACAGTCTAATTTTTCACCGTGCGTTTTATTGAT
GTTTTCTTCTATAATAACGTCACAGAATAAAAAGGTGACATTCCTACCCTCTAATTCTTGAACGAGTTTATGA
CAGAAGATGACTGTACCTTTTGCGGCTGAAAAATGTCTAACACATGCAGCACCTATTCCAGACGAACCTCCAA
CAAAATTCATATGGGAAAATAA
>v31.000314(123941..124255)
ATGATTAATAATAGTCCTATAAATCAGATAAAATATGATGACATGATTGTCGACAATTGTTCAATGCAAATGG
ACGTAATGGTCATGCCAAATTTATATGGTAATGTTATTGGGAATATCGGAGCTGGTATAGTTGGTGGTGCTGG
TGGTACGTCTTGTGCGGTGTTTGAATCTGGCACTCGAAATTCTGGTAGGAATCTCGTTGGGAAAAACATAGCC
ACGTCCTGTGACATGTTAGACTATTTAGGTGAT
>v31.000349(76077..76235,76277..76441,78806..78931,78963..79247)
GAAATTAAAGCTTCTTTAGAAAGTGCTCTTAATTTAGGTTACAATCACATTGATTGCGCATGGGTGTATTGCA
TTTTAGAAGACTATATTTCAAATGGTAAAATTAAACGAGAAGAATTATTCGTAACTAGCAAGGTTNNNCTATG
ACTAGTGAAAGTCGGATTTGAAGAAACACTTAGTAATCTCCGATTAGATTATTTAGATTTATATCTGATTCAT
TTTTAATATTATATCTCATTTTCATTTATTTTCATTTTAGCCTGGNNNGAAATGGAAAAACTCGTTGATTCAC
TGTCTAATTTCAATATTAGACAAATTCAAAATATTCTTGATCATTGCAAAATTAAACCAGCAATGCTTCAAGT
GGTCAATATTTATTTTCCGAATAAAAAACTAGTGGAATTTTGCCTGAAAAATGATATTCAAGTCACTGCATAT
GATCGACCTGGAGCGACAGAGGGAGAAAAACGTCCTCTTAGTGATCCAACAGTTTTAGAGTTAGCTAAAAAAT
TATTGATAAGCTATATTATTCATCGGGGTTTGATCTGCATTCCTAAGAGTGTATCCTCAGCCAGATTGGAGGA
>v31.000482(133155..133514,136192..136467)
ATGACGTTAAATTACAATTTTTCTGGAAAAACAGTTTTAGTAACAGGCGGTTCGAAAGGAATAGGAAAACAGT
CTGGAGCTAGTGTTATTTCATTAAGCAGATCTACTGGTGTATTTCAAAATTTTAGTTCAAACGGAAATAATTT
TGATATTAGTAATTGGGATGAATTTAATTCCATAATAAATTCTTTGTTGCCTGTGGATTATCTTGTTAATAAT
AAATTTGGTGACATCAATGAAAAAGAAATTGACGAAATGATCAATACAAATGTCAAATCAGTTATCAACATTT
AGGCGGCACTGGATGCTATTACACGAAATATGGCTCTTGAGCTTGGCCCGCATAACATACGGGTGAATTCTGT
AGATATGGGTCGTCTTTATTGGAATGACGAGTCTAAGAGGGATAAGTTGATTTCGAGAATTCCTTTGGCGAGA
GTCGATGTTATTATGTATTTGTTGACAGATCATTGCACGCTTGTGACGGGGGCGGCAATTCCCATTGATGGTC
new()
methods:
new()
next_seq()
Bio::SeqIO Object


seq:
methods:
new()
translate()
length()
Bio::Seq Object
data:
next_seq()
methods:
new()
next_seq()
Bio::SeqIO Object

```
#!/usr/bin/perl -w
#file: inFasta_loop.pl
use strict;
use Bio::SeqIO;

# get fasta filename from user input
my $file = shift;

# create a SeqIO obj with $file as filename
# $SeqIO_object contains all the individual sequence
# that are in file named $file
my $seqIO_object = Bio::SeqIO->new(
                        -file => $file,
                        -format => 'fasta',
                    );

# using while loop and next_seq method to "get to"
# and create a Seq obj for each individual sequence
# in the SeqIO obj of many sequences
while (my $seq_object = $seqIO_object->next_seq){
        #do stuff to each sequence in the fasta
}
```

#file: inFasta_doStuff_outFasta.pl

1. Get a file name from user input (@ARGV) and stores in $file

2. Create a new seqIO object in $seqIO_object, using filename $file and format 'fasta'

3. Create a second seqIO object in $out using format 'fasta'

4. Loop thru each seq object in $seqIO_object storing information from the object in variables.

5. Print out the stored information

6. Print out $seq_object using the method or tool 'write_seq()' and the seqIO object $out.

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my $file = shift;
my $seqIO_object = Bio::SeqIO->new(
                    -file => $file,
                    -format => 'fasta',
                                );
my $out = Bio::SeqIO->new(-format => 'fasta');

while (my $seq_object = $seqIO_object->next_seq){
        my $id = $seq_object->id;
        my $desc = $seq_object->desc;
        my $seqString = $seq_object->seq;
        my $revComp = $seq_object->revcom;
        my $alphabet = $seq_object-> alphabet;
        my $translation_seq_obj   = $seq_object-> translate;
        my $translation  = $translation_seq_obj -> seq;
        my $seqLen = $seq_object->length;

        print "translation: $translation\n";
        print "alphapet: $alphabet\n";
        print "seqLen: $seqLen\n";

       #prints to STDOUT
       $out->write_seq($seq_object);
}
```

fasta input:

```
>seqName seq description is blah blah blah
AGGCTCAATTTAGTTTTCCTTGTCCTTATTTTAAAAGGTGTCCAGTG
TGATGTGCAGCTGGTGGAGTCTGGGGGAGGCTTAGTGCAGCCTGGAG
GGTCCCGGAAACTCTCCTGTGCAGCCTCTGGATTCACTTTCAGTAGC
TTTGGAATGCACTGGGTTCGTCAGGCTCCAGAGAAGGGGCTGGAGTG
GGTCGCATACATTAGTAGTGGCAGTAGTACCCTCCACTATGCAGACA
CAGTGAAGGGCCGATTCACCATCTCAAGAGACAATCCCAAGAACACC
CTGTTCCTGCAAATGACCAGTCTAAGGTCTGAGGACACGGCCATGTA
TTACTGTGCAAGATGGGGTAACTACCCTTACTATGCTATGGACTACT
GGGGTCAA
```

output:

```
translation: RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFTFSSF
GMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSLRSEDTAM
YYCARWGNYPYYAMDYWGQGTSVTVSS
alphapet: dna
seqLen: 408
>seqName seq description is blah blah blah
AGGCTCAATTTAGTTTTCCTTGTCCTTATTTTAAAAGGTGTCCAGTGTGATGTGCAGCTG
GTGGAGTCTGGGGGAGGCTTAGTGCAGCCTGGAGGGTCCCGGAAACTCTCCTGTGCAGCC
TCTGGATTCACTTTCAGTAGCTTTGGAATGCACTGGGTTCGTCAGGCTCCAGAGAAGGGG
CTGGAGTGGGTCGCATACATTAGTAGTGGCAGTAGTACCCTCCACTATGCAGACACAGTG
AAGGGCCGATTCACCATCTCAAGAGACAATCCCAAGAACACCCTGTTCCTGCAAATGACC
AGTCTAAGGTCTGAGGACACGGCCATGTATTACTGTGCAAGATGGGGTAACTACCCTTAC
TATGCTATGGACTACTGGGGTCAAGGAACCTCAGTCACCGTCTCCTCA
```

Table from http://www.bioperl.org/wiki/HOWTO:Beginners

List of seq object methods

Table 1: Sequence Object Methods

Name	Returns	Example	Note
new	Sequence object	$so = Bio::Seq->new(-seq => "MPQRAS")	create a new one, see Bio::Seq for more
seq	sequence string	$seq = $so->seq	get or set the sequence
display_id	identifier	$so->display_id("NP_123456")	get or set an identifier
primary_id	identifier	$so->primary_id(12345)	get or set an identifier
desc	description	$so->desc("Example 1")	get or set a description
accession	identifier	$acc = $so->accession	get or set an identifier
length	length, a number	$len = $so->length	get the length
alphabet	alphabet	$so->alphabet('dna')	get or set the alphabet ('dna','rna','protein')
subseq	sequence string	$string = $seq_obj->subseq(10,40)	Arguments are start and end
trunc	Sequence object	$so2 = $so1->trunc(10,40)	Arguments are start and end
revcom	Sequence object	$so2 = $so1->revcom	Reverse complement
translate	protein Sequence object	$prot_obj = $dna_obj->translate	See the Bioperl Tutorial for more
species	Species object	$species_obj = $so->species	See Bio::Species for more

#file: inFasta_outGenBank.pl

Change ‘format’ in the new() method from ‘fasta’ to ‘genbank’ to change the way the SeqIO object $out is displayed in STDOUT.

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my $file = shift;
my $seqIO_object = Bio::SeqIO->new(
                                -file => $file,
                                -format => ‘fasta’,
                                );

my $out = Bio::SeqIO->new(-format => ‘genbank’);

while (my $seq_object = $seqIO_object->next_seq){
      $out->write_seq($seq_object); #prints to STDOUT
}
```

```
LOCUS       seqName                  408 bp    dna     linear   UNK
DEFINITION  seq description is blah blah blah
ACCESSION   unknown
FEATURES             Location/Qualifiers
BASE COUNT       95 a     98 c    111 g    104 t
ORIGIN
        1 aggctcaatt tagttttcct tgtccttatt ttaaaaggtg tccagtgtga tgtgcagctg
       61 gtggagtctg ggggaggctt agtgcagcct ggagggtccc ggaaactctc ctgtgcagcc
      121 tctggattca ctttcagtag ctttggaatg cactgggttc gtcaggctcc agagaagggg
      181 ctggagtggg tcgcatacat tagtagtggc agtagtaccc tccactatgc agacacagtg
      241 aagggccgat tcaccatctc aagagacaat cccaagaaca ccctgttcct gcaaatgacc
      301 agtctaaggt ctgaggacac ggccatgtat tactgtgcaa gatggggtaa ctacccttac
      361 tatgctatgg actactgggg tcaaggaacc tcagtcaccg tctcctca
//
```

Query a local fasta file

Query a local fasta file

You have a fasta file that contains many records.

You want to retrieve a specific record.

You do not want to loop through all records until you find the correct record.

Use Bio::DB::Fasta.

main links
- Main Page
- Getting Started
- Downloads
- Installation
- Recent changes
- Random page

documentation
- Quick Start
- FAQ
- HOWTOs
- API Docs
- Scrapbook
- BioPerl Tutorial
- Tutorials
- Deobfuscator
- Browse Modules

community
- News
- Mailing lists
- Supporting BioPerl
- BioPerl Media

Deobfuscator

Contents [hide]

What is the Deobfuscator?

The Deobfuscator was written to make it easier to determine the methods that are available from a given BioPerl module (a common BioPerl FAQ).

BioPerl is a highly object-oriented software package, with often multiple levels of inheritance. Although each individual module is usually well-documented for the methods specific to it, identifying the inherited methods is less straightforward.

The Deobfuscator indexes all of the BioPerl POD documentation, taking account of the inheritance tree (thanks to Class::Inspector), and then presents all of the methods available to each module through a searchable web interface.

Where can I find the Deobfuscator?

The Deobfuscator is currently available here, indexing *bioperl-live*.

Welcome to the BioPerl Deobfuscator [bioperl-live]

Search **class names** by string or Perl regex (examples: Bio::SeqIO, seq, fasta$)

fasta  Submit Query

OR select a class from the list:

Bio::AlignIO::fasta	fasta MSA Sequence input/output stream
Bio::AlignIO::largemultifasta	Largemultifasta MSA Sequence input/output stream
Bio::AlignIO::metafasta	Metafasta MSA Sequence input/output stream
Bio::DB::Fasta 	Fast indexed access to a directory of fasta files
Bio::DB::Flat::BDB::fasta	fasta adaptor for Open-bio standard BDB-indexed flat file
Bio::Index::Fasta	Interface for indexing (multiple) fasta files
Bio::Search::HSP::FastaHSP	HSP object for FASTA specific data
Bio::Search::Hit::Fasta	Hit object specific for Fasta-generated hits
Bio::SearchIO::fasta	A SearchIO parser for FASTA results
Bio::Seq::SeqFastaSpeedFactory	Instantiates a new Bio::PrimarySeqI (or derived class) through a factory

sort by method

Bio::AlignIO::metafasta	Metafasta MSA Sequence input/output stream
Bio::DB::Fasta	Fast indexed access to a directory of fasta files
Bio::DB::Flat::BDB::fasta	fasta adaptor for Open-bio standard BDB-indexed flat file
Bio::Index::Fasta	Interface for indexing (multiple) fasta files
Bio::Search::HSP::FastaHSP	HSP object for FASTA specific data
Bio::Search::Hit::Fasta	Hit object specific for Fasta-generated hits
Bio::SearchIO::fasta	A SearchIO parser for FASTA results
Bio::Seq::SeqFastaSpeedFactory	Instantiates a new Bio::PrimarySeqI (or derived class) through a factory

sort by method

methods for **Bio::DB::Fasta**

Method	Class	Returns	Usage
alphabet	Bio::DB::Fasta	not documented	not documented
basename	Bio::DB::Fasta	not documented	not documented
calculate_offsets	Bio::DB::Fasta	not documented	not documented
caloffset	Bio::DB::Fasta	not documented	not documented
carp	Bio::Root::RootI	not documented	not documented
CLEAR	Bio::DB::Fasta	not documented	not documented
confess	Bio::Root::RootI	not documented	not documented
dbmargs	Bio::DB::Fasta	not documented	not documented
debug	Bio::Root::Root	none	$obj->debug("This is debugging output");

Bio::DB Fasta

Other packages in the module: Bio::DB::Fasta Bio::PrimarySeq::Fasta

Toolbar

WebCvs

Summary

Bio::DB::Fasta -- Fast indexed access to a directory of fasta files

Package variables

No package variables defined.

Included modules

AnyDBM_File
Fcntl
File::Basename qw (basename dirname)
IO::File

Inherit

Bio::DB::SeqI Bio::Root::Root

Synopsis

```
use Bio::DB::Fasta;

# create database from directory of fasta files
my $db      = Bio::DB::Fasta->new('/path/to/fasta/files');
```

Can also find these pages at http://doc.bioperl.org/bioperl-live/

Bio::DB::fasta module synopsis

doc.bioperl.org

Synopsis

```
use Bio::DB::Fasta;

# create database from directory of fasta files
my $db      = Bio::DB::Fasta->new('/path/to/fasta/files');

# simple access (for those without Bioperl)
my $seq      = $db->seq('CHROMOSOME_I',4_000_000 => 4_100_000);
my $revseq   = $db->seq('CHROMOSOME_I',4_100_000 => 4_000_000);
my @ids      = $db->ids;
my $length   = $db->length('CHROMOSOME_I');
my $alphabet = $db->alphabet('CHROMOSOME_I');
my $header   = $db->header('CHROMOSOME_I');

# Bioperl-style access
my $db      = Bio::DB::Fasta->new('/path/to/fasta/files');

my $obj     = $db->get_Seq_by_id('CHROMOSOME_I');
my $seq     = $obj->seq; # sequence string
my $subseq  = $obj->subseq(4_000_000 => 4_100_000); # string
my $trunc   = $obj->trunc(4_000_000 => 4_100_000); # seq object
my $length  = $obj->length;
# (etc)

# Bio::SeqIO-style access
my $stream  = Bio::DB::Fasta->new('/path/to/files')->get_PrimarySeq_stream;
while (my $seq = $stream->next_seq) {
  # Bio::PrimarySeqI stuff
}

my $fh = Bio::DB::Fasta->newFh('/path/to/fasta/files');
```

Bio::DB::fasta module description

doc.bioperl.org

Description

Bio::DB::Fasta provides indexed access to one or more Fasta files. It provides random access to each sequence entry, and to subsequences within each entry, allowing you to retrieve portions of very large sequences without bringing the entire sequence into memory.

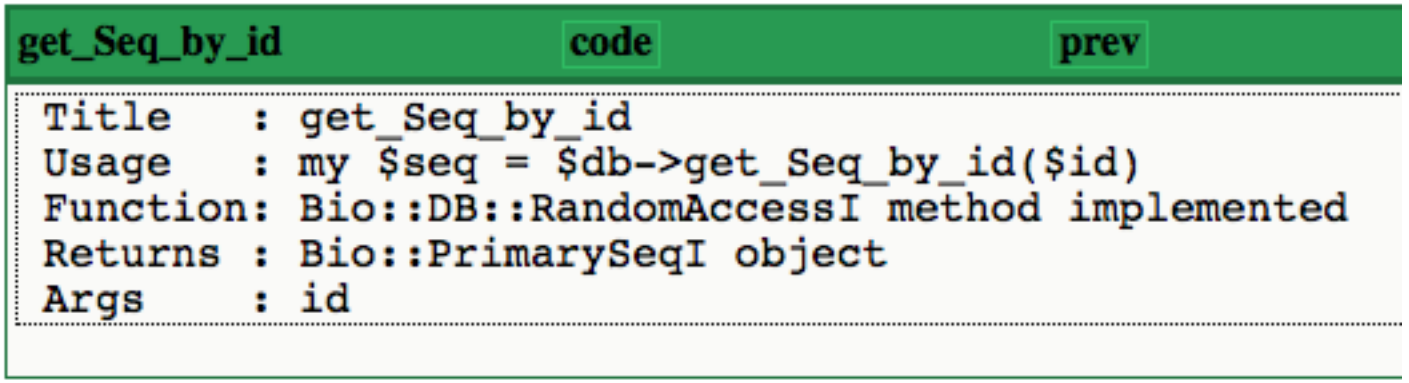
When you initialize the module, you point it at a single fasta file or a directory of multiple such files. The first time it is run, the module generates an index of the contents of the file or directory using the AnyDBM module (Berkeley DB* preferred, followed by GDBM_File, NDBM_File, and SDBM_File). Thereafter it uses the index file to find the file and offset for any requested sequence. If one of the source fasta files is updated, the module reindexes just that one file. (You can also force reindexing manually). For improved performance, the module keeps a cache of open filehandles, closing less-recently used ones when the cache is full.

The fasta files may contain any combination of nucleotide and protein sequences; during indexing the module guesses the molecular type.

Entries may have any line length up to 65,536 characters, and different line lengths are allowed in the same file. However, within a sequence entry, all lines must be the same length except for the last.

Bio::DB::fasta method description

doc.bioperl.org



#file:local_seq_query.pl

Query a local fasta file

```
#!/usr/bin/perl -w
use strict;
use Bio::DB::Fasta;

my $dbfile = 'uniprot_sprot.fasta';
my $db = Bio::DB::Fasta->new($dbfile);

# retrieve a sequence
my $id = 'sp|Q13547|HDAC1_HUMAN';
my $seq_obj = $db->get_Seq_by_id($id);

if ( $seq_obj ) {
        print "seq: ",$seq_obj->seq,"\n";
} else {
        warn("Cannot find $id\n");
}
```

output

```
seq: MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIRMTHNLLLNYGLYRKMEIYRPHKANAE
EMTKYHSDDYIKFLRSIRPDNMSEYSKQMQRFNVGEDCPVFDGLFEFCQLSTGGSVASAVKLNKQQT
DIAVNWAGGLHHAKKSEASGFCYVNDIVLAILELLKYHQRVLYIDIDIHHGDGVEEAFYTTDRVMTV
SFHKYGEYFPGTGDLRDIGAGKGKYYAVNYPLRDGIDDESYEAIFKPVMSKVMEMFQPSAVVLQCGS
DSLSGDRLGCFNLTIKGHAKCVEFVKSFNLPMLMLGGGGYTIRNVARCWTYETAVALDTEIPNELPY
NDYFEYFGPDFKLHISPSNMTNQNTNEYLEKIKQRLFENLRMLPHAPGVQMQAIPEDAIPEESGDED
EDDPDKRISICSSDKRIACEEEFSDSEEEGEGGRKNSSNFKKAKRVKTEDEKEKDPEEKKEVTEEEK
TKEEKPEAKGVKEEVKLA
```

Query a remote database

Query a remote database

You do not have a fasta file that contains the needed records.

You want to retrieve a specific record or set of records.

Records are available through GenBank.

Use Bio::DB::GenBank.

Query a remote database: GenBank with an ID (acc, gi, or unique identifier)

#file:getSeqRecord_genbank.pl

1. Use Bio::DB::Genbank module.

2. Create new SeqIO object ($out).

3. Create new GenBank DB object ($dbh).

4. Use get_Seq_by_id method.

5. Print $out (Seq object in genbank format) to STDOUT.

```
#!/usr/bin/perl -w
use strict;
use Bio::DB::GenBank;

my $out = Bio::SeqIO->new(-format => 'genbank');
my $dbh = Bio::DB::GenBank->new;
my $query = 'MUSIGHBA1';
my $seq_obj = $dbh->get_Seq_by_id($query);

if( $seq_obj ) {
        $out->write_seq($seq_obj);
} else {
        warn("cannot find sequence $query\n");
}
```

can change from 'genbank' to 'fasta' or other format

Output: GenBank file

```
LOCUS       MUSIGHBA1               408 bp    mRNA    linear   ROD 27-APR-1993
DEFINITION  Mouse Ig active H-chain V-region from MOPC21, subgroup VH-II,
            mRNA.
ACCESSION   J00522
VERSION     J00522.1  GI:195052
KEYWORDS    constant region; immunoglobulin heavy chain; processed gene; variable region; variable region sub-
group VH-II.
SOURCE      Mus musculus (house mouse).
  ORGANISM  Mus musculus
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia;
            Sciurognathi; Muroidea; Muridae; Murinae; Mus.
REFERENCE   1  (bases 1 to 408)
  AUTHORS   Bothwell,A.L., Paskind,M., Reth,M., Imanishi-Kari,T., Rajewsky,K.
            and Baltimore,D.
  TITLE     Heavy chain variable region contribution to the NPb family of
            antibodies: somatic mutation evident in a gamma 2a variable region
  JOURNAL   Cell 24 (3), 625-637 (1981)
   PUBMED   6788376
COMMENT     Original source text: Mouse C57Bl/6 myeloma MOPC21, cDNA to mRNA,
            clone pAB-gamma-1-4. [1] studies the response in C57Bl/6 mice to
            NP proteins. It is called the b-NP response because this mouse
            strain carries the b-IgH haplotype. See other entries for b-NP
            response for more comments.
FEATURES             Location/Qualifiers
     source          1..408
                     /db_xref="taxon:10090"
                     /mol_type="mRNA"
                     /organism="Mus musculus"
     CDS             <1..>408
                     /db_xref="GI:195055"
                     /codon_start=1
                     /protein_id="AAD15290.1"
                     /translation="RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFT
                     FSSFGMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSL
                     RSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS"
                     /note="Ig H-chain V-region from MOPC21"
     sig_peptide     <1..48
     mat_peptide     49..>408
                     /product="Ig H-chain V-region from MOPC21 mature peptide"
     misc_recomb     343..344
                     /note="V-region end/D-region start (+/- 1bp)"
     misc_recomb     356..357
                     /note="D-region end/J-region start"
BASE COUNT       95 a     98 c    111 g    104 t
ORIGIN      57 bp upstream of PvuII site, chromosome 12.
        1 aggctcaatt tagttttcct tgtccttatt ttaaaaggtg tccagtgtga tgtgcagctg
       61 gtggagtctg ggggaggctt agtgcagcct ggagggtccc ggaaactctc ctgtgcagcc
      121 tctggattca ctttcagtag ctttggaatg cactgggttc gtcaggctcc agagaagggg
      181 ctggagtggg tcgcatacat tagtagtggc agtagtaccc tccactatgc agacacagtg
      241 aagggccgat tcaccatctc aagagacaat cccaagaaca ccctgttcct gcaaatgacc
      301 agtctaaggt ctgaggacac ggccatgtat tactgtgcaa gatggggtaa ctacccttac
      361 tatgctatgg actactgggg tcaaggaacc tcagtcaccg tctcctca
//
```

Creating a sequence record

Creating a sequence record

You have a sequence and want to create a Seq object on the fly.

Use Bio::Seq.

Create a sequence record on the fly.

#file:createSeqOnFly.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::Seq;
use Bio::SeqIO;

my $seqObj = Bio::Seq->new(-seq => 'ATGAATGATGAA',
                           -display_id => 'seq_example',
                           -description=> 'this seq is awesome');

my $out = Bio::SeqIO->new(-format => 'fasta');
$out->write_seq($seqObj);


print "Id: ",$seqObj->display_id, "\n";
print "Length: ", $seqObj->length, "\n";
print "Seq: ",$seqObj->seq,"\n";
print "Subseq (3..6): ", $seqObj->subseq(3,6), "\n";
print "Translation: ", $seqObj->translate->seq, "\n";
```

1. Create a new seq object

2. Create and print a new seqIO object in fasta format using $seqObj

3. Get features of $seqObj by using seqObj methods

Notice the coupling of methods.

Output

```
>seq_example this seq is awesome
ATGAATGATGAA
Id: seq_example
Length: 12
Seq: ATGAATGATGAA
Subseq (3..6): GAAT
Translation: MNDE
```

File format conversions

File format conversions

You have GenBank files and want to extract only the sequence in fasta format.

Use Bio::SeqIO.

Formats

BioPerl's SeqIO system understands lot of formats and can interconvert all of them. Here is a current listing of formats, as of version 1.5.

Table 1: Bio::SeqIO modules and formats supported

Name	Description	File extension	Module
abi	ABI tracefile	ab[i1]	Bio::SeqIO::abi
ace	Ace database	ace	Bio::SeqIO::ace
agave	AGAVE XML		Bio::SeqIO::agave
alf	ALF tracefile	alf	Bio::SeqIO::alf
asciitree	write-only, to visualize features		Bio::SeqIO::asciitree
bsml	BSML, using XML::DOM	bsml	Bio::SeqIO::bsml
bsml_sax	BSML, using XML::SAX		Bio::SeqIO::bsml_sax
chadoxml	CHADO sequence format		Bio::SeqIO::chadoxml
chaos	CHAOS sequence format		Bio::SeqIO::chaos
chaosxml	Chaos XML		Bio::SeqIO::chaosxml
ctf	CTF tracefile	ctf	Bio::SeqIO::ctf

http://www.bioperl.org/wiki/HOWTO:SeqIO

```
LOCUS       MUSIGHBA1               408 bp    mRNA    linear   ROD 27-APR-1993
DEFINITION  Mouse Ig active H-chain V-region from MOPC21, subgroup VH-II,
            mRNA.
ACCESSION   J00522
VERSION     J00522.1  GI:195052
KEYWORDS    constant region; immunoglobulin heavy chain; processed gene; variable re-
gion; variable region subgroup VH-II.
SOURCE      Mus musculus (house mouse).
  ORGANISM  Mus musculus
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia;
            Sciurognathi; Muroidea; Muridae; Murinae; Mus.
REFERENCE   1  (bases 1 to 408)
  AUTHORS   Bothwell,A.L., Paskind,M., Reth,M., Imanishi-Kari,T., Rajewsky,K.
            and Baltimore,D.
  TITLE     Heavy chain variable region contribution to the NPb family of
            antibodies: somatic mutation evident in a gamma 2a variable region
  JOURNAL   Cell 24 (3), 625-637 (1981)
   PUBMED   6788376
COMMENT     Original source text: Mouse C57Bl/6 myeloma MOPC21, cDNA to mRNA,
            clone pAB-gamma-1-4. [1] studies the response in C57Bl/6 mice to
            NP proteins. It is called the b-NP response because this mouse
            strain carries the b-IgH haplotype. See other entries for b-NP
            response for more comments.
FEATURES             Location/Qualifiers
     source          1..408
                     /db_xref="taxon:10090"
                     /mol_type="mRNA"
                     /organism="Mus musculus"
     CDS             <1..>408
                     /db_xref="GI:195055"
                     /codon_start=1
                     /protein_id="AAD15290.1"
                     /translation="RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFT
                     FSSFGMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSL
                     RSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS"
                     /note="Ig H-chain V-region from MOPC21"
     sig_peptide     <1..48
     mat_peptide     49..>408
                     /product="Ig H-chain V-region from MOPC21 mature peptide"
     misc_recomb     343..344
                     /note="V-region end/D-region start (+/- 1bp)"
     misc_recomb     356..357
                     /note="D-region end/J-region start"
BASE COUNT       95 a     98 c    111 g    104 t
ORIGIN      57 bp upstream of PvuII site, chromosome 12.
        1 aggctcaatt tagttttcct tgtccttatt ttaaaaggtg tccagtgtga tgtgcagctg
       61 gtggagtctg ggggaggctt agtgcagcct ggagggtccc ggaaactctc ctgtgcagcc
      121 tctggattca ctttcagtag ctttggaatg cactgggttc gtcaggctcc agagaagggg
      181 ctggagtggg tcgcatacat tagtagtggc agtagtaccc tccactatgc agacacagtg
      241 aagggccgat tcaccatctc aagagacaat cccaagaaca ccctgttcct gcaaatgacc
      301 agtctaaggt ctgaggacac ggccatgtat tactgtgcaa gatggggtaa ctacccttac
      361 tatgctatgg actactgggg tcaaggaacc tcagtcaccg tctcctca
//
```

= GenBank Format

Fasta Format

```
>MUSIGHBA1 Mouse Ig active H-chain V-region from MOPC21,
subgroup VH-II, mRNA.
AGGCTCAATTTAGTTTTCCTTGTCCTTATTTTAAAAGGTGTCCAGTGTGATGTGCAGCTG
GTGGAGTCTGGGGGAGGCTTAGTGCAGCCTGGAGGGTCCCGGAAACTCTCCTGTGCAGCC
TCTGGATTCACTTTCAGTAGCTTTGGAATGCACTGGGTTCGTCAGGCTCCAGAGAAGGGG
CTGGAGTGGGTCGCATACATTAGTAGTGGCAGTAGTACCCTCCACTATGCAGACACAGTG
AAGGGCCGATTCACCATCTCAAGAGACAATCCCAAGAACACCCTGTTCCTGCAAATGACC
AGTCTAAGGTCTGAGGACACGGCCATGTATTACTGTGCAAGATGGGGTAACTACCCTTAC
TATGCTATGGACTACTGGGGTCAAGGAACCTCAGTCACCGTCTCCTCA
```

Convert from GenBank to fasta.

#file:convert_genbank2fasta.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my ($informat,$outformat) = ('genbank','fasta');
my ($infile,$outfile) = @ARGV;

my $in = Bio::SeqIO->new(
                -format => $informat,
                -file => $infile,
                );
my $out = Bio::SeqIO->new(
                -format => $outformat,
                -file => ">$outfile"
                );

while ( my $seqObj = $in->next_seq ) {
        $out->write_seq($seqObj);
}
```

Retrieving annotations

Retrieving annotations

You have GenBank files and want to retrieve annotations.

Use Bio::SeqIO.

Sample GenBank file with Features/Annotations

```
LOCUS       MUSIGHBA1                408 bp    mRNA    linear   ROD 27-APR-1993
DEFINITION  Mouse Ig active H-chain V-region from MOPC21, subgroup VH-II,
            mRNA.
ACCESSION   J00522
VERSION     J00522.1  GI:195052
KEYWORDS    constant region; immunoglobulin heavy chain; processed gene; variable re-
gion; variable region subgroup VH-II.
SOURCE      Mus musculus (house mouse).
  ORGANISM  Mus musculus
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia;
            Sciurognathi; Muroidea; Muridae; Murinae; Mus.
REFERENCE   1  (bases 1 to 408)
  AUTHORS   Bothwell,A.L., Paskind,M., Reth,M., Imanishi-Kari,T., Rajewsky,K.
            and Baltimore,D.
  TITLE     Heavy chain variable region contribution to the NPb family of
            antibodies: somatic mutation evident in a gamma 2a variable region
  JOURNAL   Cell 24 (3), 625-637 (1981)
   PUBMED   6788376
COMMENT     Original source text: Mouse C57Bl/6 myeloma MOPC21, cDNA to mRNA,
            clone pAB-gamma-1-4. [1] studies the response in C57Bl/6 mice to
            NP proteins. It is called the b-NP response because this mouse
            strain carries the b-IgH haplotype. See other entries for b-NP
            response for more comments
FEATURES             Location/Qualifiers
     source          1..408
                     /db_xref="taxon:10090"
                     /mol_type="mRNA"
                     /organism="Mus musculus"
     CDS             <1..>408
                     /db_xref="GI:195055"
                     /codon_start=1
                     /protein_id="AAD15290.1"
                     /translation="RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFT
                     FSSFGMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSL
                     RSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS"
                     /note="Ig H-chain V-region from MOPC21"
     sig_peptide     <1..48
     mat_peptide     49..>408
                     /product="Ig H-chain V-region from MOPC21 mature peptide"
     misc_recomb     343..344
                     /note="V-region end/D-region start (+/- 1bp)"
     misc_recomb     356..357
                     /note="D-region end/J-region start"
BASE COUNT       95 a     98 c    111 g    104 t
ORIGIN      57 bp upstream of PvuII site, chromosome 12.
        1 aggctcaatt tagttttcct tgtccttatt ttaaaaggtg tccagtgtga tgtgcagctg
       61 gtggagtctg ggggaggctt agtgcagcct ggagggtccc ggaaactctc ctgtgcagcc
      121 tctggattca ctttcagtag ctttggaatg cactgggttc gtcaggctcc agagaagggg
      181 ctggagtggg tcgcatacat tagtagtggc agtagtaccc tccactatgc agacacagtg
      241 aagggccgat tcaccatctc aagagacaat cccaagaaca ccctgttcct gcaaatgacc
      301 agtctaaggt ctgaggacac ggccatgtat tactgtgcaa gatggggtaa ctacccttac
      361 tatgctatgg actactgggg tcaaggaacc tcagtcaccg tctcctca
//
```

```
FEATURES             Location/Qualifiers
     source          1..408
                     /db_xref="taxon:10090"
                     /mol_type="mRNA"
                     /organism="Mus musculus"
     CDS             <1..>408
                     /db_xref="GI:195055"
                     /codon_start=1
                     /protein_id="AAD15290.1"
                     /translation="RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFT
                     FSSFGMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSL
                     RSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS"
                     /note="Ig H-chain V-region from MOPC21"
     sig_peptide     <1..48
     mat_peptide     49..>408
                     /product="Ig H-chain V-region from MOPC21 mature peptide"
     misc_recomb     343..344
                     /note="V-region end/D-region start (+/- 1bp)"
     misc_recomb     356..357
                     /note="D-region end/J-region start"
```

primary_tag

tag=value

Get annotations from a GenBank file

#file: get_annot_from_genbank.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;

my $infile = shift;
my $seqIO = Bio::SeqIO->new(
        -file => $infile,
        -format => 'genbank',
    );
while (my $seqObj = $seqIO -> next_seq){
    my $name = $seqObj -> id;
    foreach my $feature ($seqObj->get_SeqFeatures){
        my $primary_tag = $feature->primary_tag;
        my ($start, $end) = ($feature->start , $feature->end);
        my $range = $start . ".." . $end;
        foreach my $tag ( sort $feature->get_all_tags ) {
            my @values = $feature->get_tag_values($tag);
            my $value_str = join ",", @values;
            print "$name($range)\t$primary_tag\t$tag:$value_str\n";
        }
    }
}
```

get_SeqFeature produces an array of Bio::SeqFeatureI objects

Output

```
MUSIGHBA1(1..408)       source  db_xref:taxon:10090
MUSIGHBA1(1..408)       source  mol_type:mRNA
MUSIGHBA1(1..408)       source  organism:Mus musculus
MUSIGHBA1(1..408)       CDS     codon_start:1
MUSIGHBA1(1..408)       CDS     db_xref:GI:195055
MUSIGHBA1(1..408)       CDS     note:Ig H-chain V-region from MOPC21
MUSIGHBA1(1..408)       CDS     protein_id:AAD15290.1
MUSIGHBA1(1..408)       CDS     translation:RLNLVFLVLILKGVQCDVQLVESGGGLVQPGGSRKLSCAASGFTFSSF
GMHWVRQAPEKGLEWVAYISSGSSTLHYADTVKGRFTISRDNPKNTLFLQMTSLRSEDTAMYYCARWGNYPYYAMDYWGQGTSVTVSS
MUSIGHBA1(49..408)      mat_peptide     product:Ig H-chain V-region from MOPC21 mature pep-
tide
MUSIGHBA1(343..344)     misc_recomb     note:V-region end/D-region start (+/- 1bp)
MUSIGHBA1(356..357)     misc_recomb     note:D-region end/J-region start
```

Bioperl II

Sofia Robb
University of Utah

BLAST

Multiple Alignments

Other cool things

BLAST

Parsing BLAST reports

Running BLAST

Parsing BLAST reports

Use Bio::SearchIO

Where do you start?

Here's an example of how one would use SearchIO to extract data from a BLAST report:

```
use Bio::SearchIO;
$report_obj = new Bio::SearchIO(-format => 'blast',
                                -file   => 'report.bls');
while( $result = $report_obj->next_result ) {
    while( $hit = $result->next_hit ) {
      while( $hsp = $hit->next_hsp ) {
         if ( $hsp->percent_identity > 75 ) {
           print "Hit\t", $hit->name, "\n", "Length\t", $hsp->length('total'),
                 "\n", "Percent_id\t", $hsp->percent_identity, "\n";
         }
       }
     }
}
```

HOWTO:SearchIO

Abstract

This is a HOWTO about the Bio::SearchIO system, how to use it, and how one goes about writing new adaptors to different output formats. We will also describe how the Bio::SearchIO::Writer modules work for outputting various formats from Bio::Search objects.

NCBI BLAST Report

Result

```
BLASTX 2.2.12 [Aug-07-2005]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.

Query= smed-HDAC1-1
         (1213 letters)

Database: swissprot.aa
           427,028 sequences; 157,875,145 total letters

Searching.................................................done
```

Hit

```
                                                                 Score     E
Sequences producing significant alignments:                     (bits) Value

sp|P56517|HDAC1_CHICK RecName: Full=Histone deacetylase 1; Short...   535   e-151
```

HSP

```
>sp|P56517|HDAC1_CHICK RecName: Full=Histone deacetylase 1; Short=HD1
          Length = 480

 Score =  535 bits (1379), Expect = e-151
 Identities = 255/343 (74%), Positives = 292/343 (85%), Gaps = 1/343 (0%)
 Frame = +3

Query: 3    CPVFDGLFEFCQLSAGGSVASAVKLNKNKADIAINWSGGLHHAKKSEASGFCYVNDIVMG 182
            CPVFDGLFEFCQLSAGGSVASAVKLNK + DIA+NW+GGLHHAKKSEASGFCYVNDIV+
Sbjct: 100  CPVFDGLFEFCQLSAGGSVASAVKLNKQQTDIAVNWAGGLHHAKKSEASGFCYVNDIVLA 159

Query: 183  ILELLKYHERVLYVDIDIHHGDGVEEAFYTTDRVMTVSFHKYGEYFPXXXXXXXXXXXXX 362
            ILELLKYH+RVLY+DIDIHHGDGVEEAFYTTDRVMTVSFHKYGEYFP
Sbjct: 160  ILELLKYHQRVLYIDIDIHHGDGVEEAFYTTDRVMTVSFHKYGEYFPGTGDLRDIGAGKG 219

Query: 363  XNYAVNFPLRDGIDDESYESIFKPVVEKVIESFKPNAIVLQCGADSLSGDRLGCFNLSLK 542
              YAVN+PLRDGIDDESYE+IFKPV+ KV+E+F+P+A+VLQCG+DSLSGDRLGCFNL++K
Sbjct: 220  KYYAVNYPLRDGIDDESYEAIFKPVISKVMETFQPSAVVLQCGSDSLSGDRLGCFNLTIK 279

Query: 543  GHGKCVEYMRQQPIPLLMLGGGGYTIRNVARCWTYETALALGTTIPNELPYNDYYEYFTP 722
            GH KCVE+++   +P+LMLGGGGYTIRNVARCWTYETA+AL T IPNELPYNDY+EYF P
Sbjct: 280  GHAKCVEFVKSFNLPMLMLGGGGYTIRNVARCWTYETAVALDTEIPNELPYNDYFEYFGP 339

Query: 723  DFKLHISPSNMANQNTPEYLERMKQKLFENLRSIPHAPSVQMQDIPEDAMDIDDGEQMDN 902
            DFKLHISPSNM NQNT EYLE++KQ+LFENLR +PHAP VQMQ IPEDA+  D G++ +
Sbjct: 340  DFKLHISPSNMTNQNTNEYLEKIKQRLFENLRMLPHAPGVQMQPIPEDAVQEDSGDE-EE 398

Query: 903  ADPDKRISILASDKYREHEADLSDSEDEGD-NRKNVDCFKSKR 1028
              DP+KRISI  SDK    + + SDSEDEG+  RKNV  FK  +
Sbjct: 399  EDPEKRISIRNSDKRISCDEEFSDSEDEGEGGRKNVANFKKAK 441
```

Result

```
 Database: /common/data/swissprot.aa
    Posted date:  Oct 4, 2009  2:02 AM
  Number of letters in database: 157,875,145
  Number of sequences in database:  427,028

Lambda     K      H
   0.318    0.134    0.401

Gapped
Lambda     K      H
   0.267   0.0410    0.140


Matrix: BLOSUM62
Gap Penalties: Existence: 11, Extension: 1
Number of Hits to DB: 281,587,467
Number of Sequences: 427028
Number of extensions: 5577736
Number of successful extensions: 16223
Number of sequences better than 1.0e-10: 1
Number of HSP's better than  0.0 without gapping: 15290
Number of HSP's successfully gapped in prelim test: 0
Number of HSP's that attempted gapping in prelim test: 0
Number of HSP's gapped (non-prelim): 16078
length of database: 157,875,145
effective HSP length: 119
effective length of database: 107,058,813
effective search space used: 30404702892
frameshift window, decay const: 40,  0.1
T: 12
A: 40
X1: 16 ( 7.3 bits)
X2: 38 (14.6 bits)
X3: 64 (24.7 bits)
S1: 41 (21.7 bits)
```

Bookmark it!!

See

http://www.bioperl.org/wiki/HOWTO:SearchIO

for a GREAT example of a blast report,

code to parse it,

a table of methods,

and the values the methods return.

Bio::SearchIO object for BLAST reports

```
#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;
#file: blast_parser_intro.pl

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
                    -file => $blast_report,
                    -format => 'blast'
                   );
```

Result object and methods

#file: sample_Blast_parse_1.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
                        -file => $blast_report,
                        -format => 'blast'
                        );

while (my $result_obj = $searchIO_obj ->next_result ) {
        my $program = $result_obj ->algorithm;
        my $queryName = $result_obj ->query_name;
        my $queryDesc = $result_obj ->query_description;
        my $queryLen = $result_obj ->query_length;
        print "program=$program\tqueryName=$queryName\t";
        print "queryDesc=$queryDesc\tqueryLen=$queryLen\n";
}
```

Output:

program=BLASTX queryName=smed-HDAC1-1 queryDesc=histone deacetylase 1 queryLen=1213

http://www.bioperl.org/wiki/HOWTO:SearchIO

Object	Method	Example	Description
Result	algorithm	BLASTX	algorithm string
Result	algorithm_version	2.2.4 [Aug-26-2002]	algorithm version
Result	query_name	20521485|dbj|AP004641.2	query name
Result	query_accession	AP004641.2	query accession
Result	query_length	3059	query length
Result	query_description	Oryza sativa ... 977CE9AF checksum.	query description
Result	database_name	test.fa	database name
Result	database_letters	1291	number of residues in database
Result	database_entries	5	number of database entries
Result	available_statistics	effectivespaceused ... dbletters	statistics used
Result	available_parameters	gapext matrix allowgaps gapopen	parameters used
Result	num_hits	1	number of hits
Result	hits		List of all Bio::Search::Hit::GenericHit object(s) for this Result
Result	rewind		Reset the internal iterator that dictates where next_hit() is pointing, useful for re-iterating through the list of hits.

Hit object and methods

#file: sample_Blast_parser_2.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SearchIO;

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
                        -file => $blast_report,
                        -format => 'blast'
                        );

while (my $result_obj = $searchIO_obj->next_result ) {
        while (my $hit_obj = $result_obj->next_hit){
                my $hitName = $hit_obj->name;
                my $hitAcc = $hit_obj->accession;
                my $hitLen = $hit_obj->length;
                my $hitSig = $hit_obj->significance;
                my $hitScore = $hit_obj->raw_score;

                print "hitName=$hitName\thitAcc=$hitAcc\thitLen=$hitLen\t";
                print "hitSig=$hitSig\thitScore=$hitScore\n";
        }
}
```

must get hit objects from a result object

Output:

hitName=sp|P56517|HDAC1_CHICK hitAcc=P56517 hitLen=480 hitSig=1e-151 hitScore=535

http://www.bioperl.org/wiki/HOWTO:SearchIO

Hit	name	443893|124775	hit name
Hit	length	331	Length of the Hit sequence
Hit	accession	443893	accession (usually when this is a genbank formatted id this will be an accession number- the part after the *gb* or *emb*)
Hit	description	LaForas sequence	hit description
Hit	algorithm	BLASTX	algorithm
Hit	raw_score	92	hit raw score
Hit	significance	2e-022	hit significance
Hit	bits	92.0	hit bits
Hit	hsps		List of all Bio::Search::HSP::GenericHSP object(s) for this Hit
Hit	num_hsps	1	number of HSPs in hit
Hit	locus	124775	locus name
Hit	accession_number	443893	accession number
Hit	rewind		Resets the internal counter for next_hsp() so that the iterator will begin at the beginning of the list

HSP object and methods

#file: sample_Blast_parser.pl

```
#!/usr/bin/perl -w
use strict;

use Bio::SearchIO;

my $blast_report = shift;

my $searchIO_obj = Bio::SearchIO->new(
                        -file => $blast_report,
                        -format => 'blast'
                        );

while (my $result_obj = $searchIO_obj->next_result ) {
      while (my $hit_obj = $result_obj->next_hit){
            while (my $hsp_obj = $hit_obj ->next_hsp){
                  my $evalue = $hsp_obj->evalue;
                  my $hitString = $hsp_obj->hit_string;
                  my $queryString = $hsp_obj->query_string;
                  my $homologyString = $hsp_obj->homology_string;

                  print "hsp evalue: $evalue\n";
                  print "HIT     : ",substr($hitString,0,50),"\n";
                  print "HOMOLOGY: ",substr($homologyString,0,50),"\n";
                  print "QUERY   : ",substr($queryString,0,50),"\n";
            }
      }
}
```

must get hsp objects from a hit object

Output:

```
hsp evalue: 1e-151
HIT     : CPVFDGLFEFCQLSAGGSVASAVKLNKQQTDIAVNWAGGLHHAKKSEASG
HOMOLOGY: CPVFDGLFEFCQLSAGGSVASAVKLNK + DIA+NW+GGLHHAKKSEASG
QUERY   : CPVFDGLFEFCQLSAGGSVASAVKLNKNKADIAINWSGGLHHAKKSEASG
```

http://www.bioperl.org/wiki/HOWTO:SearchIO

HSP	algorithm	BLASTX	algorithm
HSP	evalue	2e-022	e-value
HSP	expect	2e-022	alias for evalue()
HSP	frac_identical	0.884615384615385	Fraction identical
HSP	frac_conserved	0.923076923076923	fraction conserved (conservative and identical replacements aka "fraction similar") (only valid for Protein alignments will be same as frac_identical)
HSP	gaps	2	number of gaps
HSP	query_string	DMGRCSSG ..	query string from alignment
HSP	hit_string	DIVQNSS ...	hit string from alignment
HSP	homology_string	D+ + SSGCN	string from alignment
HSP	length('total')	52	
HSP	length('hit')	50	
HSP	length('query')	15(	
HSP	hsp_length	52	
HSP	frame	0	
HSP	num_conserved	48	
HSP	num_identical	46	
HSP	rank	1	
HSP	seq_inds('query','identical')	(96	
HSP	seq_inds('query','conserved-not-identical')	(96	

HSP	seq_inds('query','conserved')	(966,967,969,971,973,974,975, ...)	conserved or identical positions as array
HSP	seq_inds('hit','identical')	(197,202,203,204,205, ...)	identical positions as array
HSP	seq_inds('hit','conserved-not-identical')	(198,200)	conserved not identical positions as array
HSP	seq_inds('hit','conserved',1)	(197,202-246)	conserved or identical positions as array, with runs of cons
HSP	score	227	score
HSP	bits	92.0	score in bits
HSP	range('query')	(2896,3051)	start and end as array
HSP	range('hit')	(197,246)	start and end as array
HSP	percent_identity	88.4615384615385	% identical
HSP	strand('hit')	1	strand of the hit
HSP	strand('query')	1	strand of the query
HSP	start('query')	2896	start position from alignment
HSP	end('query')	3051	end position from alignment
HSP	start('hit')	197	start position from alignment
HSP	end('hit')	246	end position from alignment
HSP	matches('hit')	(46,48)	number of identical and conserved as array
HSP	matches('query')	(46,48)	number of identical and conserved as array
HSP	get_aln	*sequence alignment*	Bio::SimpleAlign object
HSP	hsp_group	*Not available in this report*	Group field from WU-BLAST reports run with -topcomboN
HSP	links	*Not available in this report*	Links field from WU-BLAST reports run with -links showin

Running BLAST

Running BLAST

on the command line

with Bioperl

BLAST on the command line:

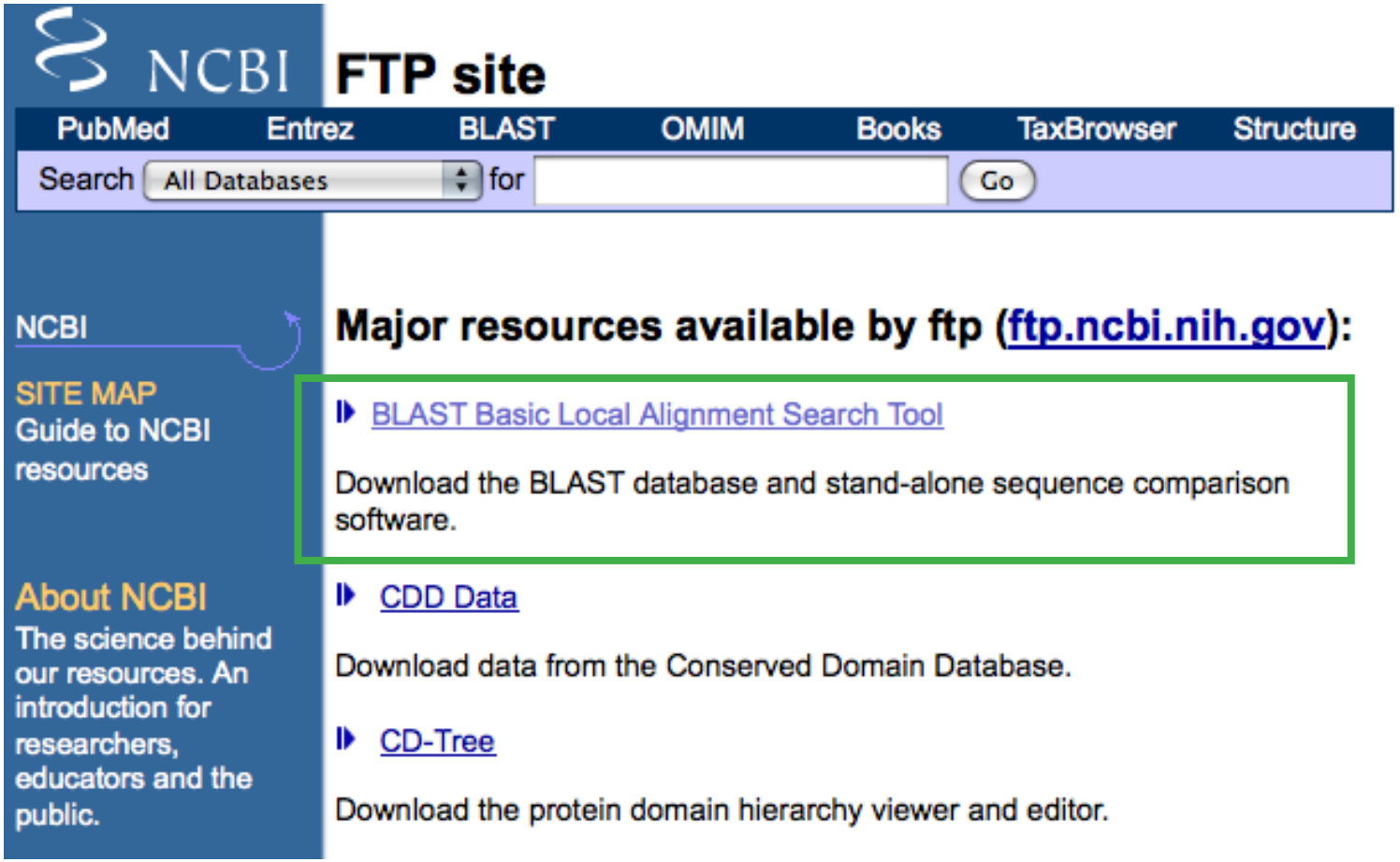
get it.

install it.

run it.

Get BLAST

http://www.ncbi.nlm.nih.gov/Ftp/



http://www.bioperl.org/wiki/HOWTO:StandAloneBlast

HOWTO:StandAloneBlast

Contents [hide]

Running BLAST on the command line

blastall -p blastn -d database -i QUERY -o out.QUERY

see ftp://ftp.ncbi.nih.gov/blast/documents/blastall.html for more information on options

formatdb

Included in the blastall package.

Used to format fasta files as blast databases.

For parameters, run the following at the command line
‘formatdb --help’.

Common usage:

formatdb -i yourFasta -p T/F -o T

Running BLAST with Bioperl

Use Bio::Tools::Run::StandAloneBlast

Getting sequences from a fasta file

```
#!/usr/bin/perl -w
#file:runblast_fasta_parse_reportObj_1.pl
use strict;

use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::SearchIO;

my $inFasta = shift;
my $seqIO_obj = Bio::SeqIO -> new (
                        -format => 'fasta',
                        -file   => $inFasta
                    );
while (my $seqObj = $seqIO_obj->next_seq){
        #run blastall with $seqObj
}
```

Setting up BLAST parameters

```
#!/usr/bin/perl -w
#file:runblast_fasta_parse_reportObj_2.pl
use strict;
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::SearchIO;

my $inFasta = shift;
my $seqIO_obj = Bio::SeqIO -> new (
                        -format => 'fasta',
                        -file   => $inFasta
                    );
#create blast parameters
my @params = (  program  => 'blastx',
            database => '/common/data/swissprot.aa',
            expect => 1e-40, b => 5, v => 5
);
while (my $seqObj = $seqIO_obj->next_seq){
      #run blastall with blast parameters
      #run blastall with $seqObj
}
```

Running BLAST

#file:runblast_fasta_parse_reportObj_3.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::SearchIO;

my $inFasta = shift;
my $seqIO_obj = Bio::SeqIO -> new (
                        -format => 'fasta',
                        -file   => $inFasta
                    );
#create blast parameters
my @params = (  program  => 'blastx',
              database => '/common/data/swissprot.aa',
              expect => 1e-40, b => 5, v => 5
);
while (my $seqObj = $seqIO_obj->next_seq){
      #set up blast object with parameters
      my $blast_obj = Bio::Tools::Run::StandAloneBlast->new(@params);
      #run blastall with blast object
      my $report_obj = $blast_obj->blastall($seqObj);
}
```

Parsing Report Object

#file: runBlast_fasta_parse_reportObj.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::SearchIO;

my $inFasta = shift;
my $seqIO_obj = Bio::SeqIO -> new (
                        -format => 'fasta',
                        -file   => $inFasta
                    );
#create blast parameters
my @params = (  program  => 'blastx',
                database => '/common/data/swissprot.aa',
                expect => 1e-40, b => 5, v => 5
);
print "QueryName\tHit_Name\tEvalue\n";
while (my $seqObj = $seqIO_obj->next_seq){
        #set up blast object with parameters
        my $blast_obj = Bio::Tools::Run::StandAloneBlast->new(@params);
        #run blastall with blast object
        my $report_obj = $blast_obj->blastall($seqObj);
        #parse report object
        while( my $result_obj = $report_obj->next_result ) {
                my $queryName = $result_obj->query_name;
                while( my $hit_obj = $result_obj->next_hit ) {
                        my  $hitName = $hit_obj->name;
                        while (my $hsp_obj = $hit_obj -> next_hsp){
                                my $hspEvalue = $hsp_obj->evalue;
                                print "$queryName\t$hitName\t$hspEvalue\n";
                        }
                }
        }
}
```

Writing BLAST results to STDOUT

#file:runblast_inFasta_write_report.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;
use Bio::SearchIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::SearchIO::Writer::TextResultWriter;

my $inFasta = shift;
my $seqIO_obj = Bio::SeqIO -> new (
                        -format => 'fasta',
                        -file   => $inFasta
                    );
while (my $seqObj = $seqIO_obj->next_seq){
      my @params = (  program  => 'blastx',
                  database => '/common/data/swissprot.aa',
                  expect => 1e-40    );
      #set up blast object with parameters
      my $blast_obj = Bio::Tools::Run::StandAloneBlast->new(@params);
      #run blastall with blast object
      my $report_obj = $blast_obj->blastall($seqObj);
      #parse report object
      print "Hit_Name\tEvalue\n";
      while( my $result_obj = $report_obj->next_result ) {
            while( my $hit_obj = $result_obj->next_hit ) {
                  print $hit_obj->name,"\t", $hit_obj->significance, "\n";
            }
      }
      #write output to STDOUT
      my $writer = Bio::SearchIO::Writer::TextResultWriter->new();
      my $out = Bio::SearchIO->new(   -writer => $writer );
      $out->write_result($report_obj->next_result);
}
```

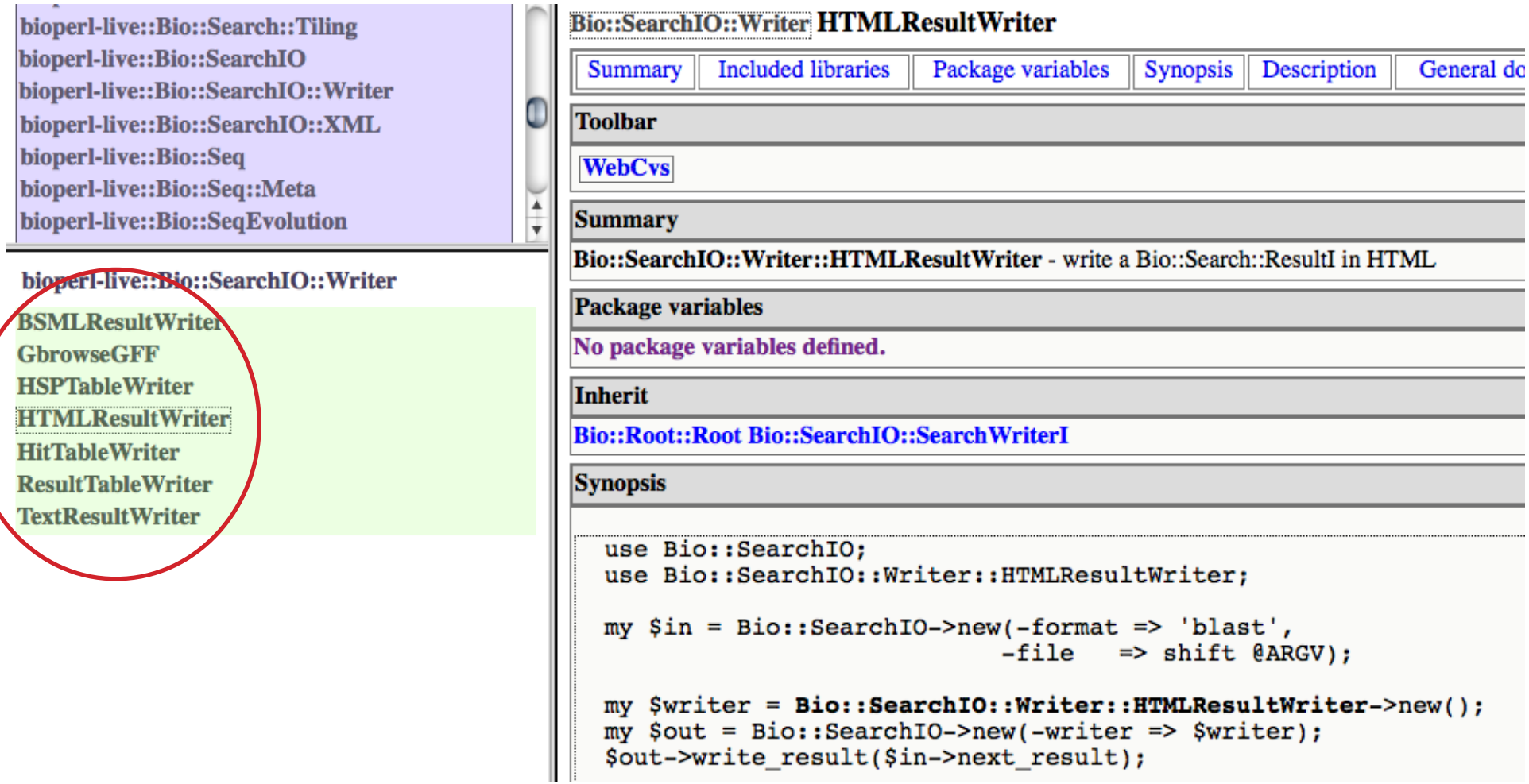
Writing BLAST results to a file

#file:runblast_inFasta_write_report.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;
use Bio::SearchIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::SearchIO::Writer::TextResultWriter;

my $inFasta = shift;
my $seqIO_obj = Bio::SeqIO -> new (
                        -format => 'fasta',
                        -file   => $inFasta
                    );
while (my $seqObj = $seqIO_obj->next_seq){
      my @params = (  program  => 'blastx',
                    database => '/common/data/swissprot.aa',
                    expect => 1e-40   );
      #set up blast object with parameters
      my $blast_obj = Bio::Tools::Run::StandAloneBlast->new(@params);
      #run blastall with blast object
      my $report_obj = $blast_obj->blastall($seqObj);
      #parse report object
      print "Hit_Name\tEvalue\n";
      while( my $result_obj = $report_obj->next_result ) {
            while( my $hit_obj = $result_obj->next_hit ) {
                  print $hit_obj->name,"\t", $hit_obj->significance, "\n";
            }
      }
      #write output to file
      my $writer = Bio::SearchIO::Writer::TextResultWriter->new();
      my $out = Bio::SearchIO->new(   -writer => $writer.
                                      -file => 'blast.out' );
      $out->write_result($report_obj->next_result);
}
```

Bio::SearchIO::Writer modules

doc.bioperl.org



www.bioperl.org/wiki/HOWTO:SearchIO

Table 1: SearchIO modules and formats supported

Name	Description
blast	BLAST (BLAST, PSIBLAST, PSITBLASTN, RPSBLAST, WUBLAST, bl2seq, WU-BLAST, BLASTZ, BLAT, Paracel BTK)
fasta	FASTA -m9 and -m0
blasttable	BLAST tabular -m9 or -m8 (NCBI) and -mformat 2 or -mformat 3 (WU-BLAST)
blastxml	NCBI BLAST XML and WU-BLAST XML
erpin	ERPIN versions 4.2.5 and above
infernal	Infernal versions 0.7 and above
megablast	MEGABLAST
psl	UCSC formats PSL
waba	WABA
axt	AXT
sim4	Sim4
hmmer	HMMER hmmpfam and hmmsearch
exonerate	Exonerate CIGAR
wise	Genewise -genesf
rnamotif	raw rnamotif output for RNAMotif versions 3.0 and above

Multiple Alignments

Use Bio::AlignIO

for parsing and writing multiple alignment file formats including:

fasta, phylip, nexus, clustalw, msf, mega, meme, pfam, psi, selex, stockholm.

Convert from fasta_aln to nexus

#file: multi_align_convert.pl

```
#!/usr/bin/perl -w
use strict;
use Bio::AlignIO;

my $align_fasta = shift;
my $in_alignIO_obj = Bio::AlignIO->new(
                        -format => 'fasta',
                        -file => $align_fasta
                        );
my $out_alignIO_obj = Bio::AlignIO->new(
                        -format => 'nexus',
                        -file => ">$align_fasta.nex"
                        );
while( my $align_obj = $in_alignIO_obj->next_aln ){
        $out_alignIO_obj->write_aln($align_obj);
}
```

next_aln produces a Bio::SimpleAlign object

Bio::SimpleAlign Object

Remove some sequences and rewrite the result

Extract or remove columns

Calculate consensus string and percent identity

Other Cool Things

Whole set of wrappers for running Bioinformatics tools in bioperl-run

Run BLAST locally or submit remote jobs (through NCBI)

Run PAML - handles setup and take down of temporary files and directories

Run alignment progs through similar interfaces: TCoffee, MUSCLE, Clustalw

Create and query databases

Relational Databases for sequence and features

Repository of scripts to do really cool things. (http://www.bioperl.org/wiki/Scripts)

Building Biological Databases

Sheldon McKay

What is a database?

Outline

- Existing databases
- Types of databases
- SQL and relational database basics
- Normalization
- Denormalization
- Dealing with very dense data
- Factors in database choice

3

Why do I need a database?

- Keeping data in flat files has limits
  - requires parsers
  - little or no organizing principles
  - inefficient and awkward to query

4

Existing Biological Databases

- Don't reinvent the wheel; has someone already solved your problem?
    - Genome annotation
        - Ensembl, Bio::DB::GFF, etc.
    - Model organism and general purpose
        - Chado
    - Maps
        - CMap, ArkDB
    - Pathways
        - Reactome, Panther, BioCyc
    - Warehousing/data mining
        - Biomart

5

Existing Biological Databases

- However there is still a need for…
    - Create lab databases, such as LIMS
    - Design databases for the special and unique data that only your group possesses
    - Extend and improve existing models and schemas to suit your needs.

6

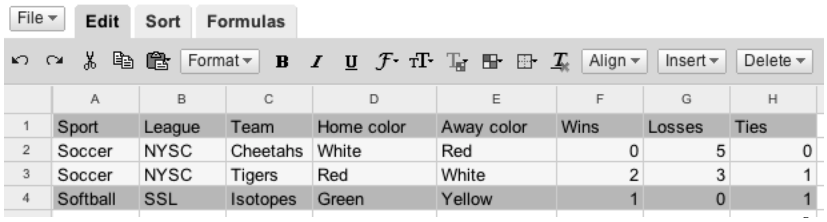
Selected Data Models

- Flat Model
- Hierarchical Model
- Object Model
- Relational Model

7

Flat (Table) Model

- A single, two-dimensional array of data elements
- All members of a column are assumed to be similar values
- All members of a row are assumed to be related to one another
- Example: spreadsheet



	A	B	C	D	E	F	G	H
1	Sport	League	Team	Home color	Away color	Wins	Losses	Ties
2	Soccer	NYSC	Cheetahs	White	Red	0	5	0
3	Soccer	NYSC	Tigers	Red	White	2	3	1
4	Softball	SSL	Isotopes	Green	Yellow	1	0	1

Hierarchical Model

- Data organized into a tree-like structure
- Represents relationships like parent/child

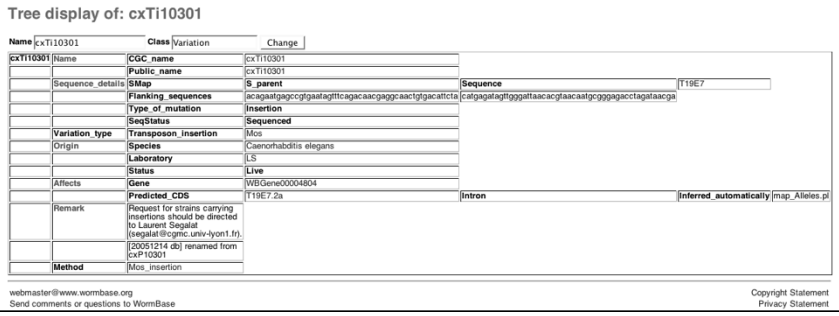
System_id	System
10	electrical
20	drive-train
30	brakes

Parent_system	part
10	ABS relay
10	flasher
10	body control module
20	transmission
20	head gasket

9

Object Model

- Attempt to apply object-oriented programming principles, such as encapsulation and polymorphism, to database organization
- Reached its peak of popularity in the 1990s
- Successfully applied to genome annotation database AceDB



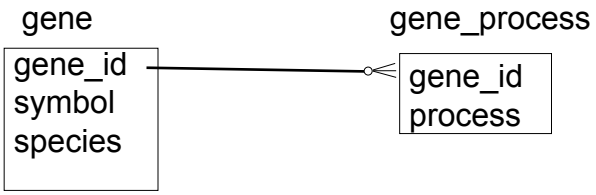
Relational Model

- A mathematical model, originally proposed by E. F. Codd in 1970, defined in terms of predicate logic and set theory
- Relational databases implement a model that is an approximation to Codd's mathematical model

11

Relational Model

- A relational database contains multiple tables, each similar to the one in the "flat" database model
- Relationships between tables are not defined explicitly; instead, keys are used to match up rows of data in different tables



12

DataBase Management Systems

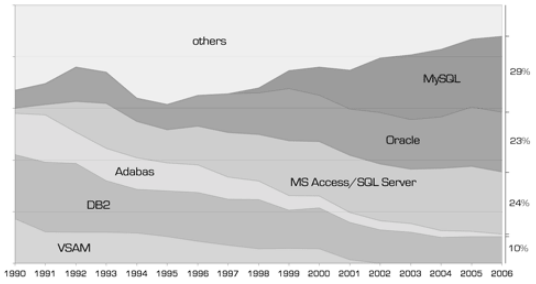
- A complex set of software programs that controls the organization, storage and retrieval of data *in a database*
- In principal, this term is not interchangeable with the database itself.

13

14

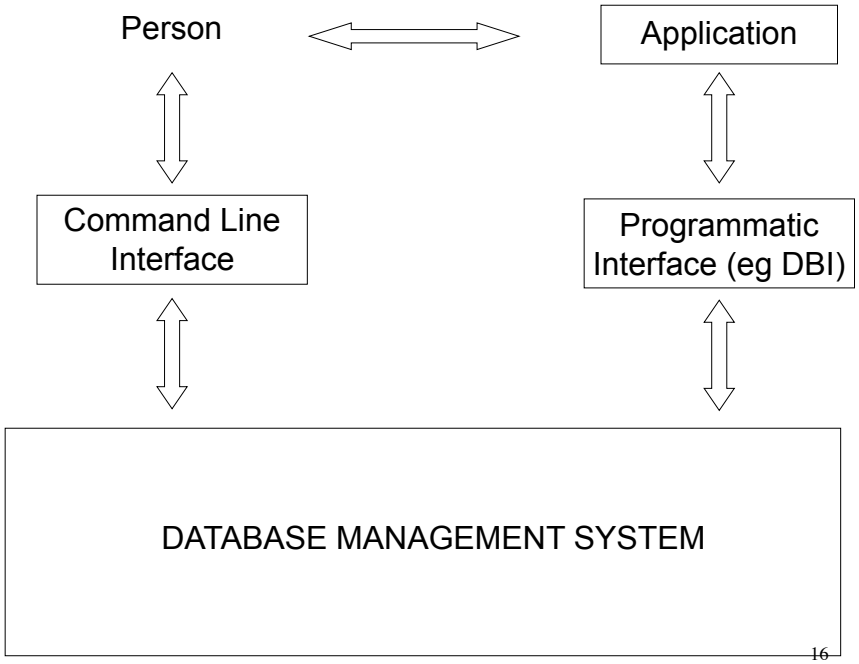
Common Relational DBMSs

- Free/open source
  - MySQL
  - PostgreSQL
- Commercial
  - Oracle
  - Sybase
  - DB2
  - Microsoft SQL Server



Source: JoinVision E-Services GmbH, July 2006

15



16

Relational database basics

- SQL
- Tables and Schemas
- Data Description Language
- Adding data
- Queries

17

Structured Query Language

• Structured English Query Language ("SEQUEL") was designed to manipulate and retrieve data stored in IBM's original relation database "System R".

• The acronym SEQUEL was later condensed to SQL due to trademark issues.

• SQL was adopted as a standard by ANSI (American National Standards Institute) in 1986.

• According to ANSI, the official pronunciation for SQL is /ɛs kjuː ɛl/ (but it also common to pronounce it *sequel).*

• Although SQL is a defined standard, there are many flavors, some proprietary

18

What can you do with SQL?

- Data retrieval
  SELECT
- Data manipulation
  INSERT, UPDATE, MERGE, TRUNCATE, DELETE
- Data definition
  CREATE, DROP
- Data transaction
  START TRANSACTION, COMMIT, ROLLBACK
- Data Control
  GRANT, REVOKE

19

Tables and queries

- Data stored in tables
  - Each row is a tuple (ordered list)
- A collection of *table definitions* is called a *schema*
- Data are entered and retrieved using SQL statements

20

A table

Table: 'favorite_color'
4 columns
2 rows (tuples)

id	first_name	last_name	color
100	Tom	Jones	blue
101	John	Smith	red

21

SQL Data Description Language

```
CREATE TABLE favorite_color (
        id              INTEGER,
        first_name      VARCHAR(255),
        last_name       VARCHAR(255),
        color           VARCHAR(255)
);
```

22

Some SQL Domains

- INTEGER
- CHAR
- VARCHAR
- FLOAT
- DOUBLE PRECISION
- TEXT
- DATE

Some constraints:

- UNIQUE
- NOT NULL

23

Adding data using SQL INSERTs

```
CREATE TABLE favorite_color (
        id              INTEGER,
        first_name      VARCHAR(255),
        last_name       VARCHAR(255),
        color           VARCHAR(255)
);

INSERT INTO favorite_color VALUES
 (100, 'Tom', 'Jones', 'blue');
INSERT INTO favorite_color VALUES
 (101, 'John', 'Smith', 'red');
```

24

Retrieving data

- Data can be retrieved using SQL queries
- SQL can used interactively, or programmatically (eg via DBI)
- SQL is based (loosely) on *relational algebra*
  - a set of operations for manipulating relations
  - main operations:
    - PROJECT
    - RESTRICT
    - JOIN

25

Projection

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	T cell differentiation

SELECT symbol FROM gene;

26

Restriction

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	T cell differentiation

```
SELECT *
FROM   gene
WHERE  species='Ovis aries'
```

27

Combining operators: restrict +project

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	T cell differentiation

```
SELECT symbol, species
FROM   gene
WHERE  process='Immune response'
```

28

Closure

- The result of a relational query is always a relation
- This allows queries to be *composed*
  - the result of a query can be used in another query

29

Closure

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	T cell differentiation

```
SELECT process
FROM   gene
WHERE  id='Q13478'
```

30

MySQL command line interface: interactive

```
[smckay@brie3 DBI_lecture]$ mysql -usmckay -pcourse genes

mysql> desc favorite_color;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | int(11)      | YES  |     | NULL    |       |
| first_name | varchar(255) | YES  |     | NULL    |       |
| last_name  | varchar(255) | YES  |     | NULL    |       |
| color      | varchar(255) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> select * from favorite_color;
+------+------------+-----------+-------+
| id   | first_name | last_name | color |
+------+------------+-----------+-------+
|  100 | Tom        | Jones     | blue  |
|  101 | John       | Smith     | red   |
+------+------------+-----------+-------+
2 rows in set (0.00 sec)
```

31

MySQL command line interface: passing SQL via STDIN

```
DROP TABLE favorite_color;
CREATE TABLE favorite_color (
        id              INTEGER,
        first_name      VARCHAR(255),
        last_name       VARCHAR(255),
        color           VARCHAR(255)
);
INSERT INTO favorite_color VALUES (100,'Tom','Jones','blue');
INSERT INTO favorite_color VALUES (101,'John','Smith','red');
SELECT * FROM favorite_color;
```

```
[smckay@brie3 DBI_lecture]$ mysql -usmckay -pcourse genes <input.txt
id      first_name      last_name       color
100     Tom     Jones   blue
101     John    Smith   red
```

32

Normalization

- process of restructuring the logical data model of a database to eliminate redundancy
- also to help organize data efficiently, and reduce the potential for anomalies during data operations

33

Normalization

Consider the data:

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	"T cell differentiation", "positive regulation of interleukin-2 biosynthesis"

A gene can belong to multiple processes. How do we model this?

34

What is wrong with this approach?

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	"T cell differentiation", "positive regulation of interleukin-2 biosynthesis"

violates *atomicity*

35

Next try: duplicating rows

id	symbol	species	process
Q13478	IL18R_Human	Homo sapiens	Immune response
P33704	CD4_CANFA	Canis familiaris	Immune response
P05542	CD4_SHEEP	Ovis aries	T cell differentiation
P05542	CD4_SHEEP	Ovis aries	positive regulation of interleukin-2 biosynthesis

problem: redundancy

36

Normalization

- Tables with redundant data are said to be not in *normal form*
- We *normalize* the schema by representing different kinds of data in different tables

37

Normalized schema

gene_id	symbol	species
Q13478	IL18R_Human	Homo sapiens
P33704	CD4_CANFA	Canis familiaris
P05542	CD4_SHEEP	Ovis aries

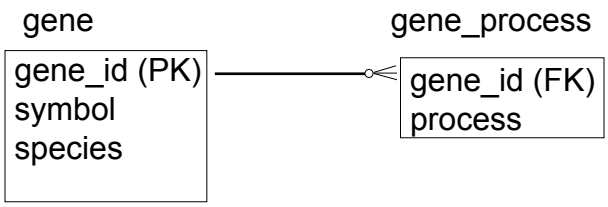
table: gene

table: gene_process

gene_id	process
Q13478	Immune response
P33704	Immune response
P05542	T cell differentiation
P05542	positive regulation of interleukin-2 biosynthesis

38

Schema



39

Primary keys and foreign keys

- A primary key for a table is one or more columns which are *guaranteed to be unique* for every row in that table
- A foreign key for a table is one or more columns that refer to a primary key in some other table

40

Choosing primary keys

- Must be unique
  - gene symbol may be a bad choice
- Primary key should be *immutable*
  - should not change during lifetime of db
- 'Natural' vs 'surrogate'
  - natural keys come from existing columns
    - Potentially useful for relating to external databases
  - surrogate keys are artificial and have no meaning
- Are database accessions 'natural'?

41

Now that we have a normalized database, how do we query across multiple tables?

- Data can be retrieved from >1 table using the JOIN operator
- The JOIN operator is actually the *composition* of two operators
  - product
  - restrict

42

Joining two tables

gene_id	symbol	species
Q13478	IL18R_Human	Homo sapiens
P33704	CD4_CANFA	Canis familiaris
P05542	CD4_SHEEP	Ovis aries

JOIN

gene_id	process
Q13478	Immune response
P33704	Immune response
P05542	T cell differentiation
P05542	positive regulation of interleukin-2 biosynthesis

gene. gene_id	gene. symbol	gene.species	gene_process. gene_id	gene_process. process
Q13478	IL18R_Human	Homo sapiens	Q13478	Immune response
P33704	CD4_CANFA	Canis familiaris	P33704	Immune response
P05542	CD4_SHEEP	Ovis aries	P05542	T cell differentiation
P05542	CD4_SHEEP	Ovis aries	P05542	positive regulation of interleukin-2 biosynthesis

43

Join syntax

gene_id	symbol	species
Q13478	IL18R_Human	Homo sapiens
P33704	CD4_CANFA	Canis familiaris
P05542	CD4_SHEEP	Ovis aries

gene_id	process
Q13478	Immune response
P33704	Immune response
P05542	T cell differentiation
P05542	positive regulation of interleukin-2 biosynthesis

```
SELECT *
FROM
  gene, gene_process
WHERE
  gene.gene_id = gene_process.gene_id
```

44

gene

gene_id	symbol	species
Q13478	IL18R_Human	Homo sapiens
P33704	CD4_CANFA	Canis familiaris
P05542	CD4_SHEEP	Ovis aries

gene_process

gene_id	process
Q13478	Immune response
P33704	Immune response
P05542	T cell differentiation
P05542	positive regulation of interleukin-2 biosynthesis

```
SELECT *
FROM
  gene, gene_process
```

Cartesian product of gene, gene_process

g.gene_id	g.symbol	g.species	gp.gene_id	gp.process
Q13478	IL18R_Human	Homo sapiens	Q13478	Immune response
P33704	CD4_CANFA	Canis familiaris	Q13478	Immune response
P05542	CD4_SHEEP	Ovis aries	Q13478	Immune response
Q13478	IL18R_Human	Homo sapiens	P33704	Immune response
P33704	CD4_CANFA	Canis familiaris	P33704	Immune response
P05542	CD4_SHEEP	Ovis aries	P33704	Immune response
Q13478	IL18R_Human	Homo sapiens	P05542	T cell differentiation
P33704	CD4_CANFA	Canis familiaris	P05542	T cell differentiation
P05542	CD4_SHEEP	Ovis aries	P05542	T cell differentiation
Q13478	IL18R_Human	Homo sapiens	P05542	positive regulation of…
P33704	CD4_CANFA	Canis familiaris	P05542	positive regulation of…
P05542	CD4_SHEEP	Ovis aries	P05542	positive regulation of…

gene

gene_id	symbol	species
Q13478	IL18R_Human	Homo sapiens
P33704	CD4_CANFA	Canis familiaris
P05542	CD4_SHEEP	Ovis aries

gene_process

gene_id	process
Q13478	Immune response
P33704	Immune response
P05542	T cell differentiation
P05542	positive regulation of interleukin-2 biosynthesis

```
SELECT *
FROM
  gene AS g, gene_process AS gp
WHERE
  gene.gene_id = gene_process.gene_id
```

g.gene_id	g.symbol	g.species	gp.gene_id	gp.process
Q13478	IL18R_Human	Homo sapiens	Q13478	Immune response
P33704	CD4_CANFA	Canis familiaris	Q13478	Immune response
P05542	CD4_SHEEP	Ovis aries	Q13478	Immune response
Q13478	IL18R_Human	Homo sapiens	P33704	Immune response
P33704	CD4_CANFA	Canis familiaris	P33704	Immune response
P05542	CD4_SHEEP	Ovis aries	P33704	Immune response
Q13478	IL18R_Human	Homo sapiens	P05542	T cell differentiation
P33704	CD4_CANFA	Canis familiaris	P05542	T cell differentiation
P05542	CD4_SHEEP	Ovis aries	P05542	T cell differentiation
Q13478	IL18R_Human	Homo sapiens	P05542	positive regulation of…
P33704	CD4_CANFA	Canis familiaris	P05542	positive regulation of…
P05542	CD4_SHEEP	Ovis aries	P05542	positive regulation of…

Further normalizations

table: gene

gene_id	symbol	species	species_common_name
Q13478	IL18R_Human	Homo sapiens	Human
P33704	CD4_CANFA	Canis familiaris	Dog
P05542	CD4_SHEEP	Ovis aries	Sheep

- not fully normalized
- non-primary key columns are dependent on each other

47

Further normalizations

gene_id	symbol	species_id
Q13478	IL18R_Human	9606
P33704	CD4_CANFA	9615
P05542	CD4_SHEEP	9940

table: gene

table: species

species_id	common_name	scientific_name
9606	human	Homo sapiens
9615	dog	Canis familiaris
9940	sheep	Ovis aries

48

More normalizations

gene_id	process
Q13478	Immune response
P33704	Immune response
P05542	T cell differentiation
P05542	positive regulation of interleukin-2 biosynthesis

gene_process

needs 'keyword' table
- controlled vocabularies
- ontologies

49

More normalization

gene_id	term_id
Q13478	6955
P33704	6955
P05542	30217
P05542	45086

gene_process

term

term_id	name
6955	Immune response
30217	T cell differentiation
45086	positive regulation of interleukin-2 biosynthesis

50

Denormalization

- The process of attempting to optimize the performance of a database by adding redundant data
- Usually proceeds from a normalized database
- Sometimes required to improve performance for large batch/data-mining queries (eg Ensembl -> BioMart)

51

Should a database schema always be normalized?

- It depends…
  - updates vs queries
  - type of queries
  - philosophical disposition
- Know when to stop normalizing
  - normalized = more tables, more joins
- Data 'warehouses'
  - e.g. BioMart

52

Examples

- Bulk data downloads
  - performance is a factor
  - Query optimized, denormalized database
- Large genomic data repository
  - Data integrity, storage efficiency
  - Normalized database
- OLAP (On Line Analytical Processing)
  - Data summaries and calculated values that are not in the parent database
  - Denormalized database with pre-computed fields

53

Summary

Relational databases are backed by theory

- powerful
- fast (usually)
- but some things are hard or difficult to express

Further reading on Normalization Theory

http://en.wikipedia.org/wiki/Database_normalization

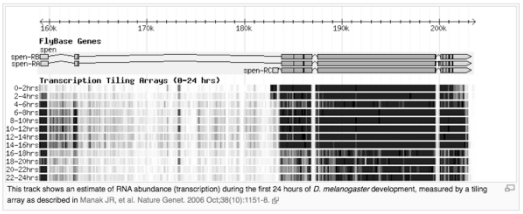
54

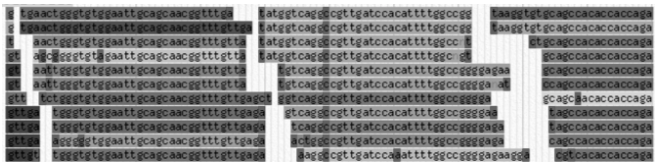
Further Reading on normalization theory

http://en.wikipedia.org/wiki/Database_normalization

Dealing with very dense data

- Microarrays
- Next-gen Sequencing





56

- Wiggle
  - Large amounts of scored data with genomic coordinates
  - Too many table rows for a relational database
  - Solution is a hybrid database/serialized data approach

WIG is a format specification introduced by the UCSC Genome Browser and also adopted by GBrowse

1) The WIG file is converted to a query-optimized binary file
2) A pointer to the binary file is stored in the database
3) An external adapter queries the binary file

http://genome.ucsc.edu/goldenPath/help/wiggle.html
http://gmod.org/wiki/GBrowse/Uploading_Wiggle_Tracks

57

- SAM/BAM (Sequence Alignment/Map)
  - NGS data generates huge numbers of aligned reads
  - The SAM specification allows efficient storage of read alignments against reference sequences
  - BAM is a highly efficient, compressed binary version of SAM
  - The SAMTools package provides utilities for handling the alignment data.
  - Third party implementers are starting to support SAM/BAM, for example Bio::DB::SAM/GBrowse

http://samtools.sourceforge.net/

58

- Choose a design that fits your data and working environment
- There are some tasks for which relational databases are not appropriate

59

- There are plenty of database schemas and tools out there
  - know which one to use and when
  - extend *vs* write your own

60

Problem Set

A quick primer on SQLite

- SQLite is a simple, stand alone, file based RDBMS

- The sqlite3 client is installed by default on many unix-like operating systems, including Mac OS X

- sqlite3 accepts two kinds of commands: meta-commands (preceded by a dot) and SQL statements.

Some useful meta-commands:

.schema tablename
.quit
.help (shows all of the available meta-commands)

Creating and loading a database:

$ sqlite3 my_database_name < my_SQL_file.sql

61

Querying databases with DBI

Sheldon McKay

Outline:

Sample data and database

SQL review and example queries

DBI
- architecture
- opening a connection
- error handling
- a basic DBI script
- basic queries
- fetch methods
- a more advanced DBI script

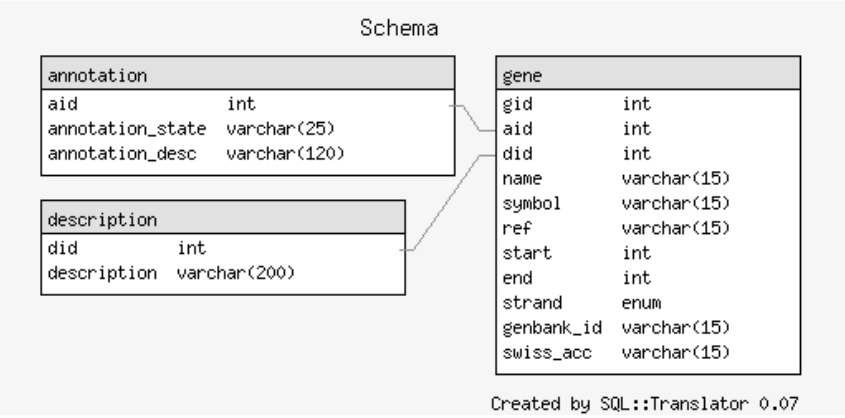
The data

Borrelia burgdorferi gene information downloaded from TIGR*

```
[smckay@brie3 dbi]$ head -20 annotations.txt |cut -f1-7
TIGR Locus      TIGR Common Name        TIGR Gene Symbol        TIGR 5' End     TIGR 3' End
     SWISS-PROT/TrEMBL Accession     GenBank ID
BB0001  hypothetical protein            105     677     O51035  AAC66406.1
BB0002  "beta-N-acetylhexosaminidase, putative"         1796    768     O54536  AAC66400.1
BB0003  hypothetical protein            3148    1784    O51036  AAC66405.1
BB0004  phosphoglucomutase      femD    3395    5188    O51037  AAC66399.1
BB0005  tryptophanyl-tRNA synthetase    trsA    6312    5251    O51038  AAC66398.1
BB0006  conserved hypothetical integral membrane protein                7433    6309    O51039  AAC66397.1
BB0007  hypothetical protein            8315    7458    O51040  AAC66404.1
BB0008  conserved hypothetical protein          8412    9197    O51041  AAC66396.1
BB0009  hypothetical protein            9202    10206   O51042  AAC66403.1
BB0010  "holo-acyl-carrier protein synthase, putative"          10203   10577   O51043  AAC66395.1
BB0011  hypothetical protein            10581   11420   O51044  AAC66402.1
BB0012  pseudouridylate synthase I      hisT    11421   12161   P70830  AAC66394.1
BB0013  hypothetical protein            12154   12753   O51046  AAC66401.1
BB0014  primosomal protein N    priA    12746   14728   Q45032  AAC66393.1
BB0015  uridine kinase  udk     15348   14725   Q59190  AAC66392.1
BB0016  glpE protein    glpE    15722   15345   O51048  AAC66391.1
BB0017  conserved hypothetical integral membrane protein                15845   16804   O51049  AAC66414.1
BB0018  conserved hypothetical protein          17817   16807   P70870  AAC66413.1
BB0019  hypothetical protein            18304   17792   O51051  AAC66416.1
```

* Now J. Craig Venter Institute

The database schema



The ‘gene’ table

table

```
sqlite> .schema gene
CREATE TABLE gene (
  gid           INTEGER  NOT NULL primary key,
  aid           INTEGER,
  did           INTEGER,
  name          TEXT NOT NULL,
  symbol        TEXT,
  ref           TEXT NOT NULL,
  start         INTEGER  NOT NULL,
  end           INTEGER  NOT NULL,
  strand        TEXT,
  genbank_id    TEXT,
  swiss_acc     TEXT
);
```

data

	name	symbol	start	end	swiss_acc	genbank_id	ref
BB0004	phosphoglucomutase	femD	3395	5188	O51037	AAC66399.1	chromosome

The ‘gene’ table

```
sqlite> SELECT * FROM gene LIMIT 10;
gid  aid  did  name     symbol  ref         start   end     strand  genbank_id    swiss_acc
---  ---  ---  -------  ------  ----------  ------  ------  ------  ------------  ----------
1    3    1    BB0001           chromosome  105     677     +       AAC66406.1    O51035
2    1    2    BB0002           chromosome  768     1796    -       AAC66400.1    O54536
3    3    1    BB0003           chromosome  1784    3148    -       AAC66405.1    O51036
4    1    3    BB0004   femD    chromosome  3395    5188    +       AAC66399.1    O51037
5    1    4    BB0005   trsA    chromosome  5251    6312    -       AAC66398.1    O51038
6    2    5    BB0006           chromosome  6309    7433    -       AAC66397.1    O51039
7    3    1    BB0007           chromosome  7458    8315    -       AAC66404.1    O51040
8    2    6    BB0008           chromosome  8412    9197    +       AAC66396.1    O51041
9    3    1    BB0009           chromosome  9202    10206   +       AAC66403.1    O51042
10   1    7    BB0010           chromosome  10203   10577   +       AAC66395.1    O51043
```

The ‘description’ table

data

```
BB0004  phosphoglucomutase      femD    3395    5188    O51037  AAC66399.1      B31
```

table

```
sqlite> .schema description
CREATE TABLE description (
  did            INTEGER NOT NULL primary key,
  description    TEXT
);
```

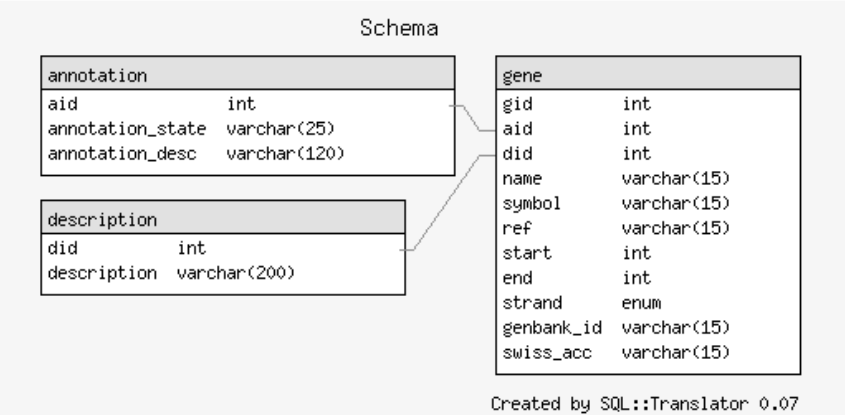
```
sqlite> SELECT * FROM description LIMIT 10;
did  description
---  --------------------------------------------------
1    hypothetical protein
2    beta-N-acetylhexosaminidase, putative
3    phosphoglucomutase
4    tryptophanyl-tRNA synthetase
5    conserved hypothetical integral membrane protein
6    conserved hypothetical protein
7    holo-acyl-carrier protein synthase, putative
8    pseudouridylate synthase I
9    primosomal protein N
10   uridine kinase
```

The ‘annotation’ table

```
sqlite> .schema annotation
CREATE TABLE annotation (
  aid               INTEGER NOT NULL primary key,
  annotation_state  TEXT NOT NULL,
  annotation_desc   TEXT
);
```

```
sqlite> SELECT * FROM annotation;
aid  annotation_state           annotation_desc
---  -------------------------  --------------------------------------------------
1    curated                    human curation; based on experimental evidence
2    conserved_hypothetical     conserved putative ORF; unknown function
3    predicted                  based on ab initio gene prediction algorithm
```

The database schema



SQL and exploration of the database

Simple queries

```
sqlite> SELECT count(*) FROM gene;
count(*)
----------
1740
```

Count the genes

```
sqlite> SELECT name,symbol,genbank_id FROM gene LIMIT 10;
name        symbol      genbank_id
----------  ----------  ----------
BB0001                  AAC66406.1
BB0002                  AAC66400.1
BB0003                  AAC66405.1
BB0004      femD        AAC66399.1
BB0005      trsA        AAC66398.1
BB0006                  AAC66397.1
BB0007                  AAC66404.1
BB0008                  AAC66396.1
BB0009                  AAC66403.1
BB0010                  AAC66395.1
```

View some data

Filtering with conditional clauses

Comparison operators:

=	Tests equality between columns and/or literal values
<>	Tests for inequality (some databases use !=, ^=, or ~=)
>,<,<=,>=	Greater than, less than, etc. (same as Perl)
IN	Tests equality of a column within a specified set of values
LIKE	Allows limited wildcard matching of strings (some databases use MATCHES or CONTAINS)

Logical operators:

AND	Returns the logical AND -- true if both sides evaluate as true
OR	Returns the logical OR -- true if either the left or right side evaluates as true
NOT	Negates the logical value of the expression that follows it

Filtering with conditional clauses

```
sqlite> SELECT name,symbol,genbank_id FROM gene WHERE symbol IS NOT NULL LIMIT 10;
name        symbol      genbank_id
----------  ----------  ----------
BB0004      femD        AAC66399.1
BB0005      trsA        AAC66398.1
BB0012      hisT        AAC66394.1
BB0014      priA        AAC66393.1
BB0015      udk         AAC66392.1
BB0016      glpE        AAC66391.1
BB0020      pfpB        AAC66412.1
BB0022      ruvB        AAC66410.1
BB0023      ruvA        AAC66409.1
BB0026      folD        AAC66407.1
sqlite> SELECT name,start,end FROM gene WHERE start > 5000 AND end < 5601;
name        start       end
----------  ----------  ----------
BBA08       5250        5585
BBE05       5377        5529
BBF11       5435        5542
BBI12       5128        5346
BBK06       5126        5236
```

Only genes with a symbol

Only genes in a coordinate range

```
sqlite> SELECT did,description FROM description
   ...> WHERE description LIKE '%protease%';
did  description
---  ---------------------------------------------------
53   periplasmic serine protease DO
65   zinc protease, putative
141  ATP-dependent protease LA
184  carboxyl-terminal protease
190  ATP-dependent Clp protease, subunit A
319  ATP-dependent Clp protease proteolytic component
320  ATP-dependent Clp protease, subunit X
390  sialoglycoprotease
420  ATP-dependent Clp protease, subunit C
```

Descriptions containing the string “protease”

Joining two tables

```
sqlite> SELECT name,symbol,description FROM gene,description
   ...> WHERE gene.did = description.did LIMIT 5;
name     symbol   description
-------  -------  -----------------------------------
BB0001            hypothetical protein
BB0002            beta-N-acetylhexosaminidase, putative
BB0003            hypothetical protein
BB0004   femD     phosphoglucomutase
BB0005   trsA     tryptophanyl-tRNA synthetase
```

More complex joins

```
sqlite> SELECT name,annotation_state,description FROM gene,annotation,description
   ...> WHERE gene.did = description.did AND gene.aid = annotation.aid
   ...> AND annotation_state <> 'predicted' LIMIT 5;
name        annotation_state      description
----------  --------------------  --------------------------------------------------
BB0002      curated               beta-N-acetylhexosaminidase, putative
BB0004      curated               phosphoglucomutase
BB0005      curated               tryptophanyl-tRNA synthetase
BB0006      conserved_hypothetical  conserved hypothetical integral membrane protein
BB0008      conserved_hypothetical  conserved hypothetical protein
```

Q: what happens if there is no description (the WHERE clause is not satisfied)?

Ordering with “ORDER BY”

```
sqlite> SELECT name,symbol,start,end FROM gene
   ...> where ref LIKE 'chromosome%' AND start > 1000 AND end < 10000
   ...> ORDER BY start DESC;
name        symbol      start       end
----------  ----------  ----------  ----------
BB0008                  8412        9197
BB0007                  7458        8315
BB0006                  6309        7433
BB0005      trsA        5251        6312
BB0004      femD        3395        5188
BB0003                  1784        3148
```

All genes on the main chromosome between positions 1001 and 10000 in descending order by ‘start’

What are the 10 longest genes in the *B. burgdorferi* genome?

```
sqlite> SELECT name,symbol,(end-start) as length FROM gene
   ...> ORDER by length DESC LIMIT 10;
name        symbol      length
----------  ----------  ----------
BBF32                   8271
BB0512                  6500
BB0420                  4484
BB0794                  4397
BB0388      rpoC        4133
BBH09                   3836
BBE02                   3833
BB0633      recB        3509
BB0579      dnaE        3485
BB0389      rpoB        3467
```

- ‘length’ is not in the database. It can be calculated and the result aliased as ‘length’

```
(end-start) as length
```

- The “ORDER BY” operation is performed on the alias value of (end-start)

```
ORDER by length DESC
```

grouping and sorting

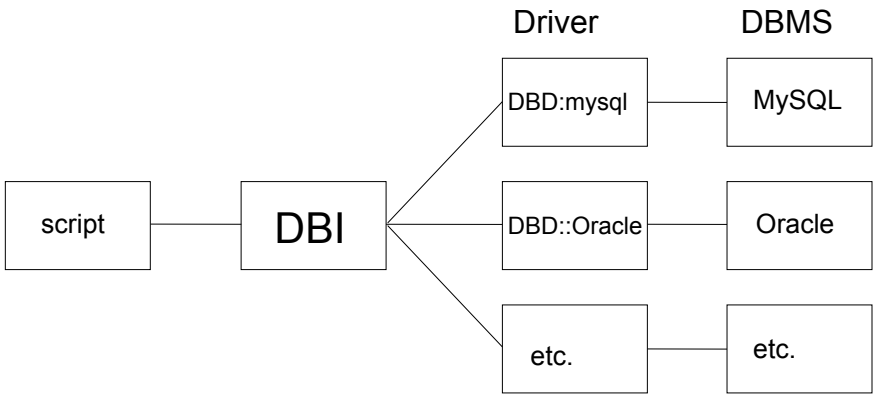
Distribution of “annotation states”

```
sqlite> SELECT annotation_state,count(*) AS number FROM gene,annotation
   ...> WHERE gene.aid = annotation.aid
   ...> GROUP by gene.aid;
annotation_state               number
-----------------------------  ----------
curated                        669
conserved_hypothetical         398
predicted                      673
```

Top 5 descriptions

```
sqlite> SELECT description,count(*) AS number FROM gene,description
   ...> WHERE gene.did = description.did
   ...> GROUP BY gene.did
   ...> ORDER BY number DESC LIMIT 5;
description                                              number
-------------------------------------------------------  ----------
hypothetical protein                                     637
conserved hypothetical protein                           327
conserved hypothetical protein, pseudogene               34
plasmid partition protein, putative                      21
conserved hypothetical integral membrane protein         18
```

DBI Architecture



How it works

DBI provides a high-level interface to a DBMS via a specific driver

The details and heavy lifting are handled by the drivers

If you know OOP and SQL, you know how to use DBI

How it works

There are two basic types of handle, a database handle and a statement handle

The database handle manages the connection and most non-query statements

The statement handle manages queries

Selected DBI class methods

(from http:/http://search.cpan.org/~timb/DBI/DBI.pm)

`connect`

```
$dbh = DBI->connect($data_source, $username, $password)
          or die $DBI::errstr;
$dbh = DBI->connect($data_source, $username, $password, \%attr)
          or die $DBI::errstr;
```

Establishes a database connection, or session, to the requested `$data_source`. Returns a database handle object if the connection succeeds. Use `$dbh->disconnect` to terminate the connection.

`available_drivers`

```
@ary = DBI->available_drivers;
@ary = DBI->available_drivers($quiet);
```

Returns a list of all available drivers by searching for `DBD::*` modules through the directories in `@INC`. By default, a warning is given if some drivers are hidden by others of the same name in earlier directories. Passing a true value for `$quiet` will inhibit the warning.

`data_sources`

```
@ary = DBI->data_sources($driver);
@ary = DBI->data_sources($driver, \%attr);
```

Returns a list of data sources (databases) available via the named driver. If `$driver` is empty or `undef`, then the value of the `DBI_DRIVER` environment variable is used.

drivers and data sources

```
#!/usr/bin/perl -w
use strict;
use DBI;

my @drivers = DBI->available_drivers;

print "\n--- Available DBI drivers and data sources ---\n\n";

for my $driver (@drivers) {
    my @sources = eval { DBI->data_sources($driver) };

    # was there an error?
    if ($@) {
        print "Something is wrong with the $driver driver\n\n";
    }
    elsif (@sources) {
        print "Driver $driver:\n\tSources: ", join("\n\t\t ",@sources), "\n\n";
    }
    else {
        print "No data sources visible for $driver\n\n";
    }
}
```

```
smckay@bush1:dbi > ./drivers_and_sources.pl

--- Available DBI drivers and data sources ---

Driver DBM:
        Sources: DBI:DBM:f_dir=.

Driver ExampleP:
        Sources: dbi:ExampleP:dir=.

Driver File:
        Sources: DBI:File:f_dir=.

Something is wrong with the Proxy driver

No data sources visible for Sponge

Driver mysql:
        Sources: DBI:mysql:RunSilent
                 DBI:mysql:boo
                 DBI:mysql:boooo
                 DBI:mysql:genes
                 DBI:mysql:mysql
                 DBI:mysql:test
                 DBI:mysql:toximoron
```

Connecting to the database

```
#!/usr/bin/perl -w
use strict;
use DBI;

my $db = 'genes';

my $dbh = DBI->connect("dbi:SQLite:$db");

print "We have a connection to $db\n";

sleep 1;

# might as well get into the habit now
$dbh->disconnect;

print "We have disconnected\n";
```

```
smckay@bush1:dbi > ./connect.pl
We have a connection to genes
We have disconnected
```

Object Methods

`prepare`

```
$sth = $dbh->prepare($statement)          or die $dbh->errstr;
$sth = $dbh->prepare($statement, \%attr)  or die $dbh->errstr;
```

Prepares a statement for later execution by the database engine and returns a reference to a statement handle object.

`execute`

```
$rv = $sth->execute               or die $sth->errstr;
$rv = $sth->execute(@bind_values) or die $sth->errstr;
```

Perform whatever processing is necessary to execute the prepared statement. An `undef` is returned if an error occurs. A successful `execute` always returns true regardless of the number of rows affected, even if it's zero (see below). It is always important to check the return status of `execute` (and most other DBI methods) for errors if you're not using "RaiseError".

`fetchrow_array`

```
@ary = $sth->fetchrow_array;
```

An alternative to `fetchrow_arrayref`. Fetches the next row of data and returns it as a list containing the field values. Null fields are returned as `undef` values in the list.

`do`

```
$rows = $dbh->do($statement)           or die $dbh->errstr;
$rows = $dbh->do($statement, \%attr)   or die $dbh->errstr;
$rows = $dbh->do($statement, \%attr, @bind_values) or die ...
```

Prepare and execute a single statement. Returns the number of rows affected or `undef` on error. A return value of `-1` means the number of rows is not known, not applicable, or not available.

`finish`

```
$rc  = $sth->finish;
```

Indicate that no more data will be fetched from this statement handle before it is either executed again or destroyed. The `finish` method is rarely needed, and frequently overused, but can sometimes be helpful in a few very specific situations to allow the server to free up resources (such as sort buffers).

`disconnect`

```
$rc = $dbh->disconnect  or warn $dbh->errstr;
```

Disconnects the database from the database handle. `disconnect` is typically only used before exiting the program. The handle is of little use after disconnecting.

A basic DBI script

```
#!/usr/bin/perl -w
use strict;
use DBI;

my $db = 'genes';
my $dbh = DBI->connect("dbi:SQLite:$db");

my $sth = $dbh->prepare(<<END);
SELECT name,symbol,description FROM gene,description
WHERE  gene.did = description.did
AND symbol IS NOT NULL
LIMIT 10
END
;

$sth->execute;
while (my @array = $sth->fetchrow_array) {
    print join("\t",@array), "\n";
}

$sth->finish;
undef $sth; # not usually necessary
$dbh->disconnect;
```

```
smckay@bush1:dbi > ./fetchrow_array.pl
BB0004  femD    phosphoglucomutase
BB0005  trsA    tryptophanyl-tRNA synthetase
BB0012  hisT    pseudouridylate synthase I
BB0014  priA    primosomal protein N
BB0015  udk     uridine kinase
BB0016  glpE    glpE protein
BB0020  pfpB    pyrophosphate--fructose 6-phosphate 1-phosphotransferase, beta subunit
BB0022  ruvB    Holliday junction DNA helicase
BB0023  ruvA    Holliday junction DNA helicase
BB0026  folD    methylenetetrahydrofolate dehydrogenase
```

Error handling

Errors can be handled by DBI automatically

Error handling can be turned on/off with the attributes RaiseError and PrintError

```
# eg
$dbh->{RaiseError} = 1
$dbh->{PrintError} = 1
```

Error messages can be accessed by:

```
# eg
print $DBI::errstr
or die $dbh->errstr
or warn $sth->errstr
```

Testing your SQL query

Are you getting what you expect?

dump_results

```
$rows = $sth->dump_results($maxlen, $lsep, $fsep, $fh);
```

Fetches all the rows from `$sth`, calls `DBI::neat_list` for each row, and prints the results to `$fh` (defaults to `STDOUT`) separated by `$lsep` (default `"\n"`). `$fsep` defaults to `", "` and `$maxlen` defaults to 35.

```
$sth->execute;
$sth->dump_results(80); # max field length of 80 chars
```

```
[smckay@brie3 dbi]$ ./dump.pl | head
'BB0004', 'femD', 'phosphoglucomutase'
'BB0005', 'trsA', 'tryptophanyl-tRNA synthetase'
'BB0012', 'hisT', 'pseudouridylate synthase I'
'BB0014', 'priA', 'primosomal protein N'
'BB0015', 'udk', 'uridine kinase'
'BB0016', 'glpE', 'glpE protein'
'BB0020', 'pfpB', 'pyrophosphate--fructose 6-phosphate 1-phosphotransferase, beta subunit'
'BB0022', 'ruvB', 'Holliday junction DNA helicase'
'BB0023', 'ruvA', 'Holliday junction DNA helicase'
'BB0026', 'folD', 'methylenetetrahydrofolate dehydrogenase'
```

Fetching Rows

`fetchrow_arrayref`

```
$ary_ref = $sth->fetchrow_arrayref;
$ary_ref = $sth->fetch;    # alias
```

Fetches the next row of data and returns a reference to an array holding the field values. Null fields are returned as `undef` values in the array. This is the fastest way to fetch data, particularly if used with `$sth->bind_columns`.

```
my $ar1 = $sth->fetchrow_arrayref;
print "The data structure:\n",Dumper($ar1),"\n";

# process the results
print join("\t",@$ar1), "\n";
while (my $ar = $sth->fetchrow_arrayref) {
    print join("\t",@$ar), "\n";
}
```

```
[smckay@brie3 dbi]$ ./fetchrow_arrayref.pl
The data structure:
$VAR1 = [
          'BB0004',
          'femD',
          'phosphoglucomutase'
        ];

BB0004  femD    phosphoglucomutase
BB0005  trsA    tryptophanyl-tRNA synthetase
BB0012  hisT    pseudouridylate synthase I
BB0014  priA    primosomal protein N
BB0015  udk     uridine kinase
etc.
```

Faster than fetchrow_array

`fetchrow_hashref`

```
$hash_ref = $sth->fetchrow_hashref;
$hash_ref = $sth->fetchrow_hashref($name);
```

An alternative to `fetchrow_arrayref`. Fetches the next row of data and returns it as a reference to a hash containing field name and field value pairs. Null fields are returned as `undef` values in the hash.

```
$sth->execute;
my $hr1 = $sth->fetchrow_hashref; # just first row
print "The data structure for the first row:\n",
      (Dumper($hr1)),"\n";
```

```
smckay@bush1:dbi > ./fetchrow_hashref.pl
The data structure for the first row:
$VAR1 = {
          'symbol' => 'femD',
          'name' => 'BB0004',
          'description' => 'phosphoglucomutase'
        };
```

Bulk Fetching

fetchall_arrayref

```
$tbl_ary_ref = $sth->fetchall_arrayref;
$tbl_ary_ref = $sth->fetchall_arrayref( $slice );
$tbl_ary_ref = $sth->fetchall_arrayref( $slice, $max_rows  );
```

The `fetchall_arrayref` method can be used to fetch all the data to be returned from a prepared and executed statement handle. It returns a reference to an array that contains one reference per row.

```
$sth->execute;
my $table = $sth->fetchall_arrayref;
print "The data structure:\n",
       (Dumper [@{$table}[0..2]]),
       "\n";
```

```
smckay@bush1:dbi > ./fetchall_arrayref.pl
The data structure:
$VAR1 = [
          [
            'BB0004',
            'femD',
            'phosphoglucomutase'
          ],
          [
            'BB0005',
            'trsA',
            'tryptophanyl-tRNA synthetase'
          ],
          [
            'BB0012',
            'hisT',
            'pseudouridylate synthase I'
          ]
        ];
```

fetchall_hashref

```
$hash_ref = $sth->fetchall_hashref($key_field);
```

The `fetchall_hashref` method can be used to fetch all the data to be returned from a prepared and executed statement handle. It returns a reference to a hash containing a key for each distinct value of the $key_field column that was fetched For each key the corresponding value is a reference to a hash containing all the selected columns and their values, as returned by fetchrow_hashref().

```
$sth->execute;
my $result = $sth->fetchall_hashref('name');
print Dumper $result;
```

```
$VAR1 = {
          'BB0012' => {
                        'symbol' => 'hisT',
                        'name' => 'BB0012',
                        'description' => 'pseudouridylate synthase I'
                      },
          'BB0005' => {
                        'symbol' => 'trsA',
                        'name' => 'BB0005',
                        'description' => 'tryptophanyl-tRNA synthetase'
                      },
          'BB0004' => {
                        'symbol' => 'femD',
                        'name' => 'BB0004',
                        'description' => 'phosphoglucomutase'
                      }
        };
```

selectall_arrayref

```
$ary_ref = $dbh->selectall_arrayref($statement);
$ary_ref = $dbh->selectall_arrayref($statement, \%attr);
$ary_ref = $dbh->selectall_arrayref($statement, \%attr, @bind_values);
```

This utility method combines "prepare", "execute" and "fetchall_arrayref" into a single call. It returns a reference to an array containing a reference to an array for each row of data fetched.

```
my $q = 'SELECT name, symbol FROM gene WHERE symbol IS NOT NULL LIMIT 3';
my $result = $dbh->selectall_arrayref($q);
print Dumper $result;
```

```
$VAR1 = [
          [
            'BB0004',
            'femD'
          ],
          [
            'BB0005',
            'trsA'
          ],
          [
            'BB0012',
            'hisT'
          ]
        ];
```

selectall_hashref

```
$hash_ref = $dbh->selectall_hashref($statement, $key_field);
$hash_ref = $dbh->selectall_hashref($statement, $key_field, \%attr);
$hash_ref = $dbh->selectall_hashref($statement, $key_field, \%attr, @bind_values);
```

This utility method combines "prepare", "execute" and "fetchall_hashref" into a single call. It returns a reference to a hash containing one entry, at most, for each row, as returned by fetchall_hashref().

```
my $q = 'SELECT name, symbol FROM gene WHERE symbol IS NOT NULL LIMIT 3';
my $result = $dbh->selectall_hashref($q, 'name');
print Dumper $result;
```

```
smckay@bush1:dbi > ./selectall_hashref.pl
$VAR1 = {
          'BB0012' => {
                        'symbol' => 'hisT',
                        'name' => 'BB0012'
                      },
          'BB0005' => {
                        'symbol' => 'trsA',
                        'name' => 'BB0005'
                      },
          'BB0004' => {
                        'symbol' => 'femD',
                        'name' => 'BB0004'
                      }
        };
```

Improving efficiency

a more advanced DBI script -- binding and placeholders

```
my $sth = $dbh->prepare(<<END);
SELECT name,symbol,d.description,ref,start,end,strand
FROM gene g,description d
WHERE  g.did = d.did
AND d.description LIKE ?
AND start >= ? AND end   <= ?
AND ref LIKE ?
END

my @queries  = (
          [ qw/%ATPase% 1 300000 chromosome%/],
          [ qw/%kinase% 1 300000 chromosome%/]
               );

for my $q (@queries) {
    $sth->execute(@$q);
    report($q,$sth->fetchall_arrayref);
}

sub report {
    my $q   = shift;
    my $ref = shift;
    my @terms = map {qq(\"$_\")} @$q;
    print "Search terms: ",join(', ',@terms), "\n";
    print join("\t",qw/name symbol description
               ref start end strand/), "\n";
    for my $array (@$ref) {
        print join("\t",map {$_ || ''} @$array), "\n";
    }
    print "---\n\n";
}
```

```
$ ./example_scripts/binding_and_placeholders.pl
Search terms: "%ATPase%", "1", "300000", "chromosome%"
name    symbol  description     ref     start   end     strand
BB0090          V-type ATPase, subunit K, putative      chromosome Borrelia burgdorferi B31     86926   87360   -
BB0091          V-type ATPase, subunit I, putative      chromosome Borrelia burgdorferi B31     87377   89203   -
BB0092  atpD    V-type ATPase, subunit D        chromosome Borrelia burgdorferi B31     89200   89814   -
BB0093  atpB    V-type ATPase, subunit B        chromosome Borrelia burgdorferi B31     89811   91115   -
BB0094  atpA    V-type ATPase, subunit A        chromosome Borrelia burgdorferi B31     91137   92792   -
BB0096          V-type ATPase, subunit E, putative      chromosome Borrelia burgdorferi B31     93433   94035   -
---

Search terms: "%kinase%", "1", "300000", "chromosome%"
name    symbol  description     ref     start   end     strand
BB0015  udk     uridine kinase  chromosome Borrelia burgdorferi B31     14725   15348   -
BB0056  pgk     phosphoglycerate kinase chromosome Borrelia burgdorferi B31     51253   52434   -
BB0128  cmk     cytidylate kinase       chromosome Borrelia burgdorferi B31     124149  124814  -
BB0239  dck     deoxyguanosine/deoxyadenosine kinase(I) subunit 2       chromosome Borrelia burgdorferi B31
244777  245394  -
BB0241  glpK    glycerol kinase chromosome Borrelia burgdorferi B31     246597  248102  +
---
```

HTML

10.18.2010

HTML

- HyperText Markup Language
- Not a programming language
- Stored in text files (just like Perl)

A basic page

```
<html>

    <head>
       <title>My web page title</title>
    </head>

    <body>

         Your HTML content here

    </body>
</html>
```

A kosher page

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
    <title>An XHTML 1.0 Strict standard template</title>
</head>

<body>

     <p>… Your HTML content here …</p>

</body>
</html>
```

Why use web standards?

- Accessibility
  - To robots
  - To people
- Stability

<Tags />

- Most tags open and close
- Tags must be nested properly

Right

```
<strong>
    <em>
        Strong and emphasis
    </em>
</strong>
```

Wrong

```
<strong>
    <em>
        Strong and emphasis
    </strong>
</em>
```

- Some tags stand alone

 <hr />

- Some tags take attributes

```
<img alt="My dog" src="rover.gif"/>

<a href="theonion.com">The Onion</a>
```

- Elements consist of start and end tags flanking content

XHTML tags

<!-->	<! DOCTYPE>	<a>	<abbr>	<acronym>	<address>	<area />	<b>	<base />	<bdo>
<big>	<blockquote>	<body>	 	<button>	<caption>	<cite>	<code>	<col />	<colgroup>
<dd>	<del>	<dfn>		<dl>	<dt>		<fieldset>	<form>	<frame />
<frameset>	<head>	<h1> - <h6>	<hr />	<html>	<i>	<iframe>	<img />	<input />	<ins>
<kbd>	<label>	<legend>		<link />	<map>	<meta />	<noframes>	<noscript>	<object>
	<optgroup>	<option>	<p>	<param />	<pre>	<q>	<samp>	<script>	<select>
<small>			<style>	<sub>	<sup>	<table>	<tbody>	<td>	<textarea>
<tfoot>	<th>	<thead>	<title>	<tr>	<tt>		<var>		

http://www.w3schools.com/tags/

Text tags

- Heading tag

```
<h1>This is a top level heading</h1>
<h6>This is the bottom level heading</h6>
```

- Paragraph tag

```
<p>This is definitely a paragraph</p>
```

- Line break

```
This is just two lines<br />
With a hard break
```

- Emphasis and Strong

```
That's <em>exactly</em> what I mean - I am <strong>sick</strong> of
this slide
```

- Comment Tag

```
<!-- This is a comment. You won't see this on the web-->
```

Tables

```
<table border="1">
    <tr>
        <th>Column 1 heading</th>
        <th>Column 2 heading</th>
        <th>Column 3 heading</th>
    </tr>
    <tr>
        <td>Row 2, cell 1</td>
        <td colspan="2">Row 2, cell 2, also spanning Row 2, cell 3</td>
    </tr>
    <tr>
        <td rowspan="2">Row 3, cell 1, also spanning Row 4, cell 1</td>
        <td>Row 3, cell 2</td>
        <td>Row 3, cell 3</td>
    </tr>
    <tr>
        <td>Row 4, cell 2</td>
        <td>Row 4, cell 3</td>
    </tr>
</table>
```

output:

Column 1 heading	Column 2 heading	Column 3 heading
Row 2, cell 1	Row 2, cell 2, also spanning Row 2, cell 3	
Row 3, cell 1, also spanning Row 4, cell 1	Row 3, cell 2	Row 3, cell 3
	Row 4, cell 2	Row 4, cell 3

http://htmldog.com/guides/htmlintermediate/tables/

Lists

```
<ol>
  <li>First things first</li>
  <ul>
    <li>Who you know</li>
  </ul>
  <li>Not</li>
  <ul>
    <li>What you know</li>
    <li>What you can do with it</li>
  </ul>
</ol>
```

output:

1. First things first
    - Who you know
2. Not
    - What you know
    - What you can do with it

Links

- Relative

```
<a href="myDirectory/index.html">Go down a directory</a>
<a href="../index.html">Go up a directory</a>
```

- Absolute

```
<a href="/">Go to the root</a>
<a href="http://nytimes.com">Go to the NY Times</a>
```

- Anchors

```
<a href="#theEnd">Go to the end</a>
<h1 id="theEnd">This is the end</h1>
```

Images

```
<img src="images/Early.png" alt="An Appalachian Mud Squid">
```

Forms

```
<form name="input" action="html_form_submit.pl" method="post">
```

- POST vs GET

Text fields

```
<form name="input" action="handleMyForm.pl" method="get">
    First name:
    <input type="text" name="firstname" />
    <br />
    Last name:
    <input type="text" name="lastname" />
    <input type="submit" value="Submit" />
</form>
```

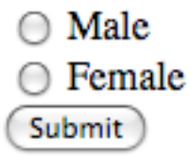
output:

First name:
Last name: Submit

Radio buttons

```
<form name="input" action="handleMyForm.pl" method="get">
    <input type="radio" name="sex" value="male"/> Male
    <br />
    <input type="radio" name="sex" value="female"/> Female
    <br />
    <input type="submit" value="Submit" />
</form>
```

output:



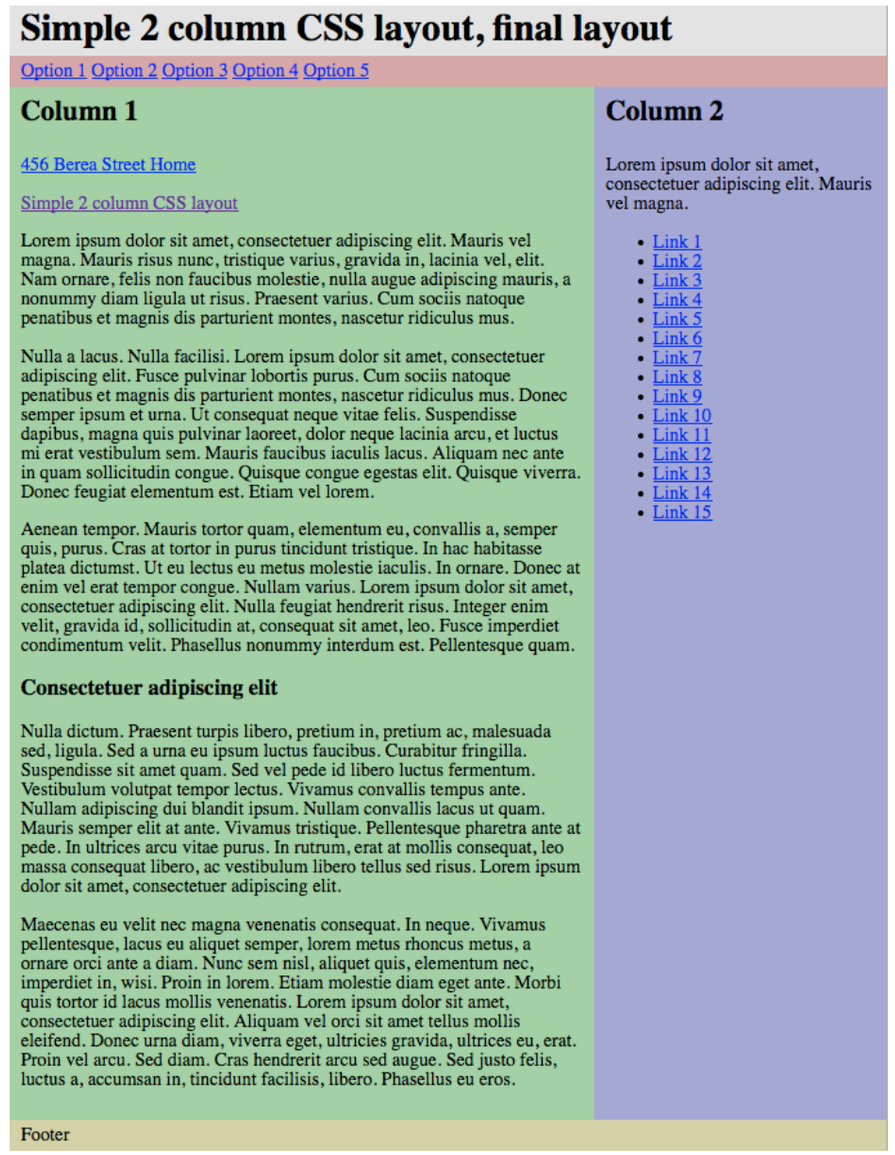
xHTML + CSS = Web

```
<body>
<div id="wrap">
    <div id="header"><h1>Simple 2 column CSS layout, final layout</h1></div>
    <div id="nav">
        <ul>
            <li><a href="#">Option 1</a></li>
            <li><a href="#">Option 2</a></li>
            <li><a href="#">Option 3</a></li>

            <li><a href="#">Option 4</a></li>
            <li><a href="#">Option 5</a></li>
        </ul>
    </div>
    <div id="main">
        <h2>Column 1</h2>
        <p><a href="/">456 Berea Street Home</a></p>

        <p><a href="/lab/developing_with_web_standards/csslayout/2-col/">Simple 2 column CSS layout<
        <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris vel magna. Mauris risus
        <p>Nulla a lacus. Nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
        <p>Aenean tempor. Mauris tortor quam, elementum eu, convallis a, semper quis, purus. Cras at
        <h3>Consectetuer adipiscing elit</h3>
        <p>Nulla dictum. Praesent turpis libero, pretium in, pretium ac, malesuada sed, ligula. Sed

        <p>Maecenas eu velit nec magna venenatis consequat. In neque. Vivamus pellentesque, lacus eu
    </div>
    <div id="sidebar">
        <h2>Column 2</h2>
        <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris vel magna.</p>
        <ul>
            <li><a href="#">Link 1</a></li>

            <li><a href="#">Link 2</a></li>
            <li><a href="#">Link 3</a></li>
            <li><a href="#">Link 4</a></li>
            <li><a href="#">Link 5</a></li>
            <li><a href="#">Link 6</a></li>
            <li><a href="#">Link 7</a></li>

            <li><a href="#">Link 8</a></li>
            <li><a href="#">Link 9</a></li>
            <li><a href="#">Link 10</a></li>
            <li><a href="#">Link 11</a></li>
            <li><a href="#">Link 12</a></li>
            <li><a href="#">Link 13</a></li>

            <li><a href="#">Link 14</a></li>
            <li><a href="#">Link 15</a></li>
        </ul>
    </div>
    <div id="footer">
        <p>Footer</p>
    </div>

</div>
</body>
```

+

```
<style type="text/css">
body,html {
    margin:0;
    padding:0;
    color:#000;
    background:#a7a09a;
}
#wrap {
    width:750px;
    margin:0 auto;
    background:#99c;
}
#header {
    padding:5px 10px;
    background:#ddd;
}
h1 {
    margin:0;
}
#nav {
    padding:5px 10px;
    background:#c99;
}
#nav ul {
    margin:0;
    padding:0;
    list-style:none;
}
#nav li {
    display:inline;
    margin:0;
    padding:0;
}
#main {
    float:left;
    width:480px;
    padding:10px;
    background:#9c9;
}
h2 {
    margin:0 0 1em;
}
#sidebar {
    float:right;
    width:230px;
    padding:10px;
    background:#99c;
}
#footer {
    clear:both;
    padding:5px 10px;
    background:#cc9;
}
#footer p {
    margin:0;
}
* html #footer {
    height:1px;
}
</style>
```

=

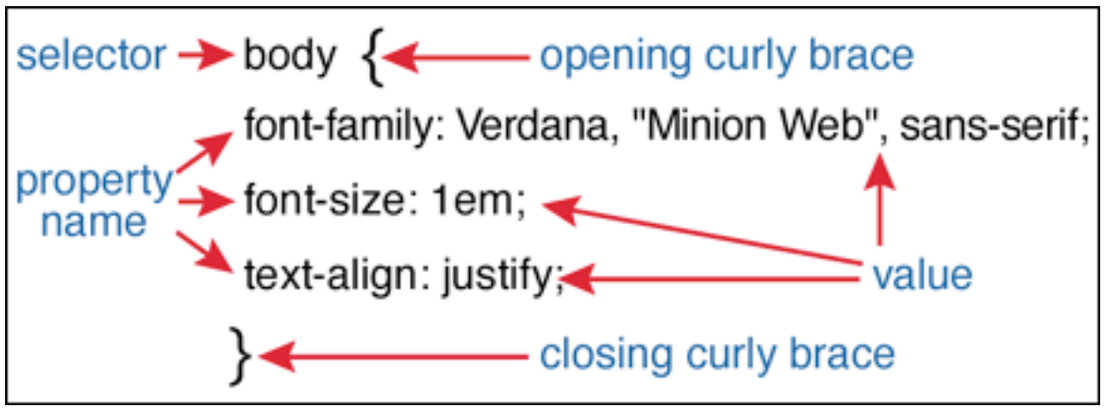


Cascading Style Sheets

- Help separate **content** from **appearance**
- One style sheet can be applied to hundreds of web pages
- Change styles in just one location

How CSS works

- Statements consist of
  - Selectors
  - Declarations
    - Properties:Values (units)



http://westciv.com/wiki/CSS_Guide:_How_do_style_sheets_work

CSS: Where do I put it?

- Embedded in the <head> of each page

```
<head><style type="text/css"> </style></head>
```

- Linked in the <head>

  Advantages: templating, speed

```
<link rel="stylesheet" type="text/css"
href="/styles/style.css" />
```

- Inline (avoid this)

```
<p style="color: red">text</p>
```

CSS Selectors

- HTML selectors - raw tags in the style sheet)
- Class selectors
  - use .className in style sheet
  - use class="className" in HTML
- ID selectors
  - use #idName in style sheet
  - use id="idName" in HTML

HTML Selector Example

Go to CNN.com
In the Firefox Web Developer Plugin change:

```
.cnn_contentarea { width:990px;text-align:left; }
```

to

```
.cnn_contentarea { width:990px;text-align:right;
color:fuchsia; letter-spacing:.15em}
```

ID Selector Example

Go to CNN.com
In the Firefox Web Developer Plugin change:

```
#cnn_maint1lftf { float:right;width:250px;margin:0px;display:inline;margin:0 0 0 5px; }
```

to

```
#cnn_maint1lftf { float:right;width:250px;margin:0px;display:inline;margin:0 0 0 5px; }
```

Class selector example

sigNotSig.html

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
     <head>
          <title>Class selectors example</title>
          <meta http-equiv="content-type"
          content="text/html;charset=utf-8" />
          <meta http-equiv="Content-Style-Type" content="text/css" />

          <link rel="stylesheet" type="text/css" href="styles/style.css" />
     </head>
     <body>
          <p class="sig">Your result is significant</p>
          <p class="notSig">Your result is not significant</p>
     </body>
</html>
```

styles/style.css

```
p.sig {
      color: green;
}

p.notSig {
      color: red;
}
```

output

Your result is significant

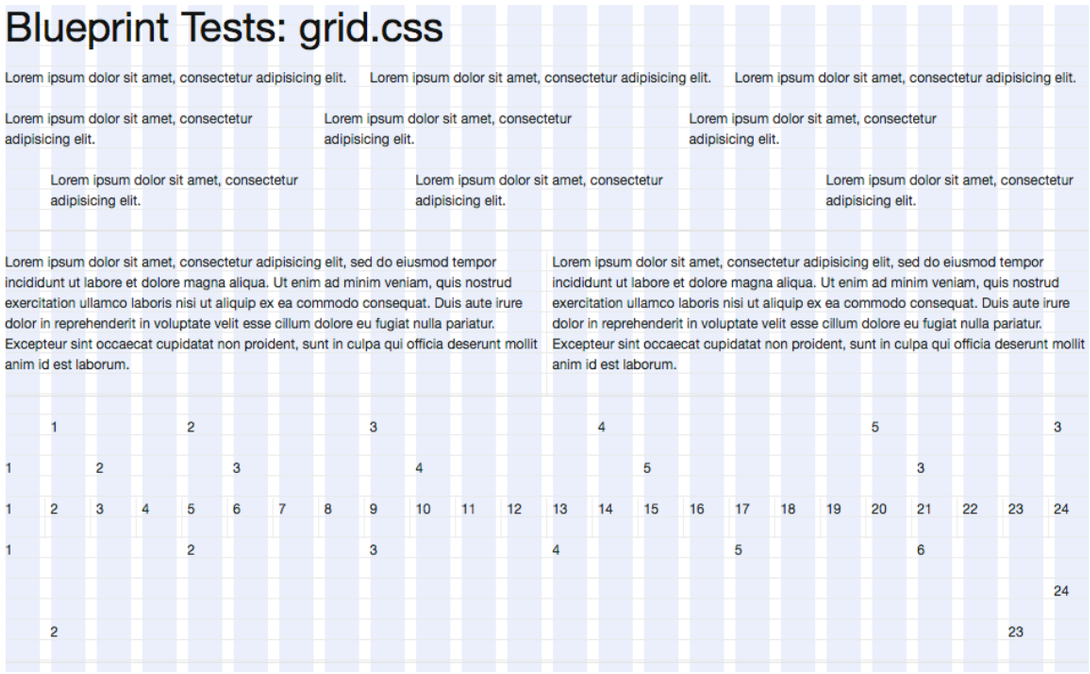
Your result is not significant

Divs and Spans

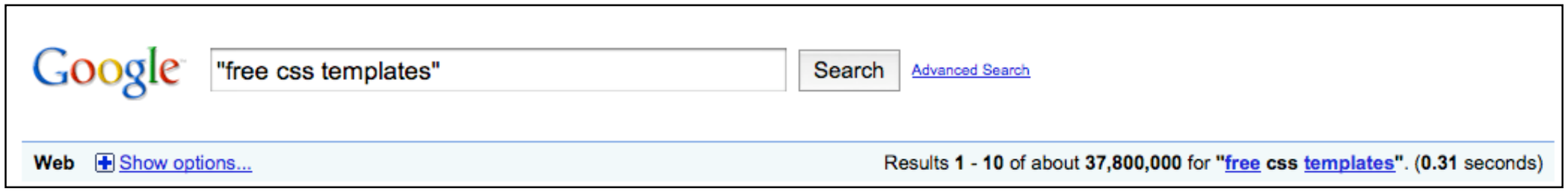
- Divs
  - Use   to define block elements. Useful for both formatting and positioning.
    - The id is unique. It refers to one element
- Spans
  - Use when you want to apply a class to some text inline
  - This is my sequence ACTGATCTAGCT

BlueprintCSS

- CSS framework
- grid
- “sensible typography”
- stylesheet for printing



Do Not Reinvent the Wheel



- http://www.freecsstemplates.org

Where does my website go?

- On Mac OS X
  - Personal web: ~/Sites
    - Main web: /Library/Webserver/Documents
- Linux: /var/www/html or /var/apache2/htdocs
- XP Home: C:\Program Files\ApacheGroup \Apache\htdocs
- Could be elsewhere. Don't give up!

Naming your html files

- .html .htm
- Why index.html is special

Where is my site?

- In this class:
  - http://infoserver.cshl.edu/~username
- On your own machine
  - http://localhost/ or http://127.0.0.1

Apache

- /etc/apache2/httpd.conf
  - The apache configuration
- /usr/sbin/apachectl
  - Apache HTTP server control interface
- /var/logs/apache2/error_log
  - Apache error log

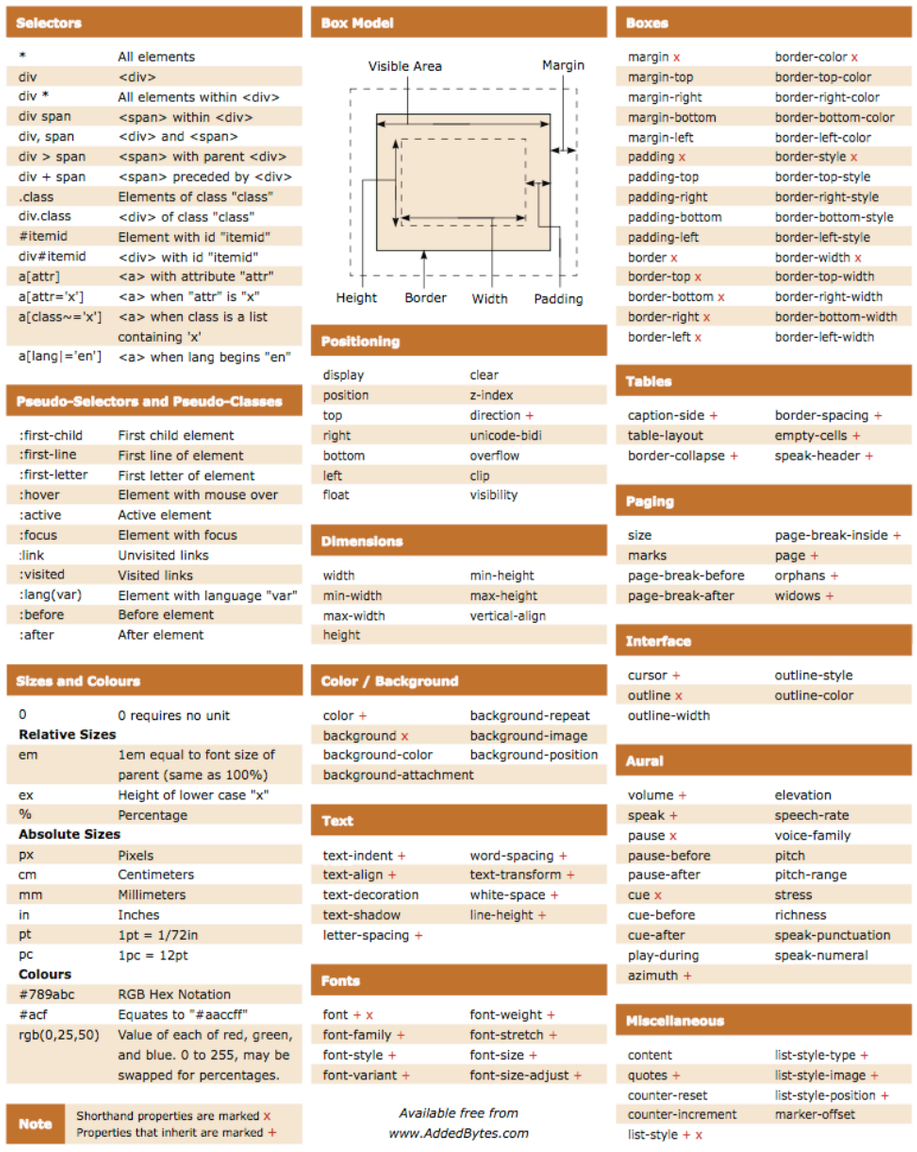
Vendor	Product	Web Sites Hosted	Percent
Apache	Apache	96,531,033	52.05%
Microsoft	IIS	61,023,474	32.90%
Google	GWS	9,864,303	5.32%
nginx	nginx	3,462,551	1.87%
lighttpd	lighttpd	2,989,416	1.61%
Oversee	Oversee	1,847,039	1.00%
Others	-	9,756,650	5.26%
Total	-	**185,474,466**	**100.00%**

Resource: HTML

- HTML Dog
  http://htmldog.com

- W3C tags
  http://www.w3schools.com/tags

Resources: CSS



Selectors

*	All elements
div	
div *	All elements within
div span	within
div, span	and
div > span	with parent
div + span	preceded by
.class	Elements of class "class"
div.class	of class "class"
#itemid	Element with id "itemid"
div#itemid	with id "itemid"
a[attr]	<a> with attribute "attr"
a[attr='x']	<a> when "attr" is "x"
a[class~='x']	<a> when class is a list containing 'x'
a[lang|='en']	<a> when lang begins "en"

Pseudo-Selectors and Pseudo-Classes

:first-child	First child element
:first-line	First line of element
:first-letter	First letter of element
:hover	Element with mouse over
:active	Active element
:focus	Element with focus
:link	Unvisited links
:visited	Visited links
:lang(var)	Element with language "var"
:before	Before element
:after	After element

Sizes and Colours

0	0 requires no unit
Relative Sizes	
em	1em equal to font size of parent (same as 100%)
ex	Height of lower case "x"
%	Percentage
Absolute Sizes	
px	Pixels
cm	Centimeters
mm	Millimeters
in	Inches
pt	1pt = 1/72in
pc	1pc = 12pt
Colours	
#789abc	RGB Hex Notation
#acf	Equates to "#aaccff"
rgb(0,25,50)	Value of each of red, green, and blue. 0 to 255, may be swapped for percentages.

Note Shorthand properties are marked x
Properties that inherit are marked +

Box Model

Positioning

display	clear
position	z-index
top	direction +
right	unicode-bidi
bottom	overflow
left	clip
float	visibility

Dimensions

width	min-height
min-width	max-height
max-width	vertical-align
height	

Color / Background

color +	background-repeat
background x	background-image
background-color	background-position
background-attachment	

Text

text-indent +	word-spacing +
text-align +	text-transform +
text-decoration	white-space +
text-shadow	line-height +
letter-spacing +	

Fonts

font + x	font-weight +
font-family +	font-stretch +
font-style +	font-size +
font-variant +	font-size-adjust +

Available free from www.AddedBytes.com

Boxes

margin x	border-color x
margin-top	border-top-color
margin-right	border-right-color
margin-bottom	border-bottom-color
margin-left	border-left-color
padding x	border-style x
padding-top	border-top-style
padding-right	border-right-style
padding-bottom	border-bottom-style
padding-left	border-left-style
border x	border-width x
border-top x	border-top-width
border-bottom x	border-right-width
border-right x	border-bottom-width
border-left x	border-left-width

Tables

caption-side +	border-spacing +
table-layout	empty-cells +
border-collapse +	speak-header +

Paging

size	page-break-inside +
marks	page +
page-break-before	orphans +
page-break-after	widows +

Interface

cursor +	outline-style
outline x	outline-color
outline-width	

Aural

volume +	elevation
speak +	speech-rate
pause x	voice-family
pause-before	pitch
pause-after	pitch-range
cue x	stress
cue-before	richness
cue-after	speak-punctuation
play-during	speak-numeral
azimuth +	

Miscellaneous

content	list-style-type +
quotes +	list-style-image +
counter-reset	list-style-position +
counter-increment	marker-offset
list-style + x	

Cheat sheet:

http://www.addedbytes.com/download/css-cheat-sheet-v2/pdf/

CSS tutorial

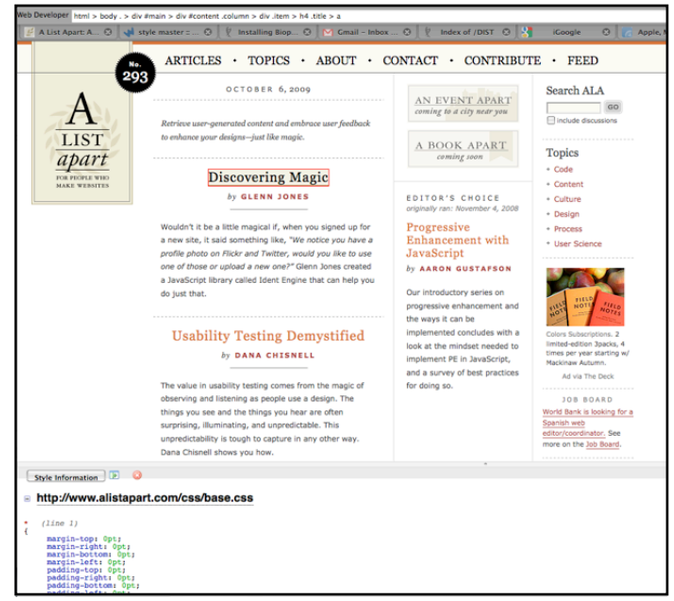
http://westciv.com/wiki/Main_Page

Two column style sheet and tutorial

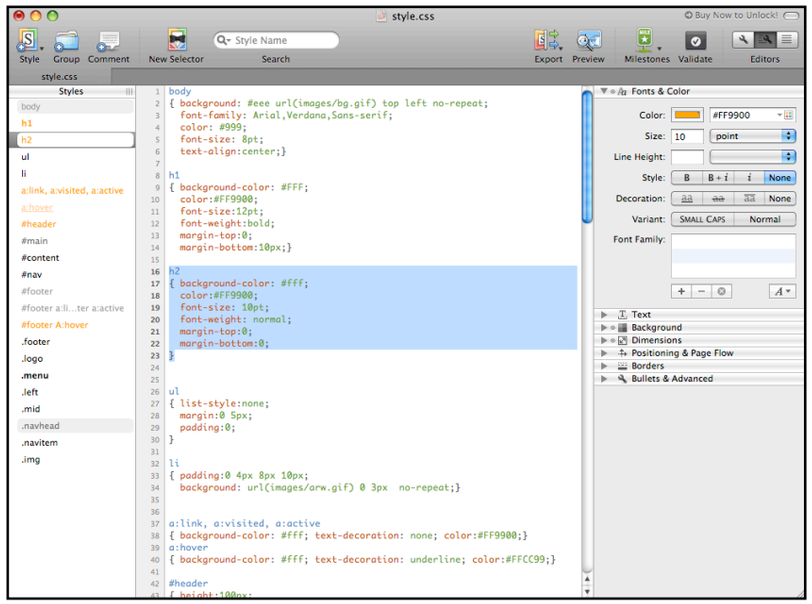
http://www.456bereastreet.com/lab/developing_with_web_standards/csslayout/2-col/

Tools of the Trade

- Web Developer Plugin for Firefox



- CSS editors
  - MacRabbit CSSEdit
  - SimpleCSS
  - TopStyle (Windows)



Web Programming with CGI.pm

Sheldon McKay

Executing CGI scripts

Use your personal web space
/Users/yourusername/Sites/cgi-bin

1) Create your script (end with '.pl')

2) `$ chmod 755 myscript.pl`

A CGI Script that Creates Plain Text

```
#!/usr/bin/perl
# file: plaintext.pl

 print "Content-type: text/plain\n\n";

 print "When that Aprill with his shoures soote\n";
 print "The droghte of March hath perced to the roote,\n";
 print "And bathed every veyne in swich licour\n";
 print "Of which vertu engendered is the flour...\n";
```

http://mckay.cshl.edu/cgi-bin/course/plaintext.pl

A CGI Script that Creates HTML

```
#!/usr/bin/perl
# file: chaucer.pl

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

 print "<html><head><title>Chaucer</title></head><body>\n";
 print "<h1>Chaucer Sez</h1>\n";

 print "When that Aprill with his shoures soote<br>\n";
 print "The droghte of March hath perced to the roote,<br>\n";
 print "And bathed every veyne in swich licour<br>\n";
 print "Of which vertu engendered is the flour...<p>\n";

 print "<cite>-Geoffrey Chaucer</cite>\n";
 print "<hr>\n";
 print "</body></html>\n";
```

http://mckay.cshl.edu/cgi-bin/course/chaucer.pl

A CGI Script that Does Something Useful

A CGI script can do anything a Perl script can do, such as opening files and processing them. Just print your results to STDOUT.

```
#!/usr/bin/perl -w
# file: process_cosmids.pl
use strict;

my @GENES   = qw/act-1 dpy-5 unc-13 let-653 skn-1 C02D5.1/;
my $URL     = 'http://www.wormbase.org/db/gene/gene?name=';

print "Content-type: text/html\n\n";
print "<html><head><title>Genes</title></head><body>\n";
print "<h1>Genes</h1>\n";
print "<ol>\n";

for my $gene (@GENES) {
  print qq(<li><a href="$URL$gene">$gene</a>\n);
}

print "</ol>\n";
print "</body></html>\n";
```

http://mckay.cshl.edu/cgi-bin/course/process_genes.pl

Creating Fill-Out Forms

HTML includes about a half-dozen elements for creating fill-out form elements. A form must begin with <FORM> and end with </FORM>:

Code:

```
<form action="http://stein.cshl.org/cgi-bin/test-cgi.pl" method="POST">
  Choose a Motif:
    <input type="text"   name="motif" value="TATTAT"> <br>
    <input type="submit" name="search" value="Search!"> <br>
</form>
```

Result:

Choose a Motif: TATTAT

Search!

Creating Fill-Out Forms II

The <FORM> Tag

Attributes:

action (required)
CGI script to submit contents of form to.

method (required)
Submission method. Depends on CGI script. One of:
- POST
- GET

encoding
Required by certain scripts that accept file uploads. One of:
- application/x-www-form-urlencoded
- multipart/form-data

Creating Fill-Out Forms III

<INPUT> Elements

Used for text fields, buttons, checkboxes, radiobuttons. Attributes:

type
Type of the field. Options:
- submit
- radio
- checkbox
- text
- password
- hidden
- file

name
Name of the field.

value
Starting value of the field. Also used as label for buttons.

size
Length of text fields.

checked
Whether checkbox/radio button is checked.

Creating Fill-Out Forms IV

Examples:

```
<input type="text"   name="motif1" value="TATTAT">

<input type="checkbox" name="motif2" value="TATTAT">

<input type="radio" name="motif3" value="TATTAT" checked>
<input type="radio" name="motif3" value="GGGGGG">

<input type="hidden" name="settings" value="PRIVACY MODE ON">

<input type="submit" name="search" value="SEARCH!">
```

Creating Fill-Out Forms V

<SELECT> Element

Used to create selection lists.

Attributes:

name
Name the field.

size
Number of options to show simultaneously.

multiple
Allow multiple options to be shown simultaneously.

<OPTION> Element

Contained within a >SELECT> element. Defines an option:

```
>option>I am an option</option>
```

Attributes:

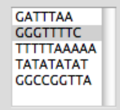
selected
Whether option is selected by default.

value
Give the option a value different from the one displayed.

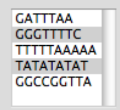
Creating Fill-Out Forms VI

```
<select name="motif1">
  <option>GATTTAA</option>
  <option selected>GGGTTTTC</option>
  <option>TTTTTAAAAA</option>
  <option>TATATATAT</option>
  <option value="Tricky!">GGCCGGTTA</option>
</select>
```


```
<select name="motif2" size=6>
  <option>GATTTAA</option>
  <option selected>GGGTTTTC</option>
  <option>TTTTTAAAAA</option>
  <option>TATATATAT</option>
  <option>GGCCGGTTA</option>
</select>
```




```
<select name="motif3" size=6 multiple>
  <option>GATTTAA</option>
  <option selected>GGGTTTTC</option>
  <option>TTTTTAAAAA</option>
  <option>TATATATAT</option>
  <option>GGCCGGTTA</option>
</select>
```




```
<input type="submit" name="search" value="SEARCH!">
```


Creating Fill-Out Forms VII

<TEXTAREA> Elements

Used to create big text elements.

Attributes:

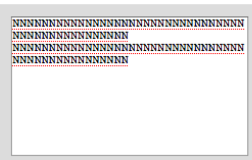
name
: name of field

rows
: rows of text

cols
: columns of text

wrap
: type of word wrapping

```
<textarea name="sequence" rows=10 cols=30>
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
</textarea>
```



```
<input type="submit" name="search" value="SEARCH!">
```

What is CGI.pm?

1. Standard module in Perl distribution (>= 5.004)
2. Emits correct HTTP headers
3. HTML shortcuts
4. Parses CGI parameters
5. "Sticky" form fields
6. Creates & processes cookies
7. File uploads

Make HTML Beautiful

CGI.pm defines functions that emit HTML. The page is easier to read and write than raw HTML*

```
<h1>
  Eat Your Vegetables
</h1>
<ol>
  <li>peas</li>
  <li>broccoli</li>
  <li>cabbage</li>
  <li>
      peppers
         <ul>
            <li>red</li>
            <li>yellow</li>
            <li>green</li>
         </ul>
    </li>
<ol>
<hr>
```

```
#!/usr/bin/perl
# Script: vegetables1.pl

use CGI ':standard';

print header,
   start_html('Vegetables'),
   h1('Eat Your Vegetables'),
   ol(
      li('peas'),
      li('broccoli'),
      li('cabbage'),
      li('peppers',
         ul(
            li('red'),
            li('yellow'),
            li('green')
            )
         ),
      ),
   hr,
   end_html;
```

* if you speak Perl!

http://mckay.cshl.edu/cgi-bin/course/vegetables.pl

Make HTML Concise

Tag Functions are Distributive

```
print li('hi','how','are','you')

    <LI>hi how are you</LI>

@items=('hi','how','are','you'); print li(\@items)

    <LI>hi</LI>
    <LI>how</LI>
    <LI>are</LI>
    <LI>you</LI>

print li(['hi','how','are','you'])

    <LI>hi</LI>
    <LI>how</LI>
    <LI>are</LI>
    <LI>you</LI>
```

Add HTML Attributes Using Hash References

```
%opts=(-type=>'square'); @items=('hi','how','are','you'); print li(\%opts,\@items)

    <LI TYPE="square">hi</LI>
    <LI TYPE="square">how</LI>
    <LI TYPE="square">are</LI>
    <LI TYPE="square">you</LI>

print li({-type=>'square'},['hi','how','are','you'])

    <LI TYPE="square">hi</LI>
    <LI TYPE="square">how</LI>
    <LI TYPE="square">are</LI>
    <LI TYPE="square">you</LI>
```

```
#!/usr/bin/perl
# Script: vegetables2.pl

use CGI ':standard';

print header,
  start_html('Vegetables'),
  h1('Eat Your Vegetables'),
  ol(
    li(['peas',
       'broccoli',
       'cabbage',
       'peppers' .
       ul(['red','yellow','green']),
       'kolrabi',
       'radishes'])
    ),
  hr,
  end_html;
```

http://mckay.cshl.edu/cgi-bin/course/vegetables2.pl

Using CGI.pm for the Genes Script

```
#!/usr/bin/perl -w
# file: process_genes2.pl

use strict;
use CGI ':standard';

my @GENES   = qw/act-1 dpy-5 unc-13 let-653 skn-1 C02D5.1/;
my $URL     = 'http://www.wormbase.org/db/gene/gene?name=';

my @list_items;
for my $gene (@GENES) {
  push @list_items,a({-href=>"$URL$gene"},$gene);
}

print header(),
    start_html('Genes'),
    h1('Genes'),
    ol(
       li(\@list_items)
       ),
    end_html;
```

http://mckay.cshl.edu/cgi-bin/course/process_genes2.pl

Setting & Retrieving CGI Parameters

You can set and retrieve CGI parameters easily. In these examples, the CGI field string is:

banana=yellow&squash=green&tomato=red&tomato=green

Retrieve a Single-Valued Field Named "Tomato":

Call *param()* with the name of the field and assign it to a scalar.

```
my $banana_color = param('banana');
# yields "yellow"
```

This works for both GET and POST types, including *multipart/form-data*.

Retrieve a Multi-Valued Field Named "Tomatoes":

Call *param()* with the name of the field and assign it to an array:

```
my @tomatoes = param('tomato');
# yields ('red','green')
```

Finding out What Parameters are Available

Call *param()* without any arguments. Will return the names of each of the parameters:

```
my @fields = param;
# yields ('banana','squash','tomato')
```

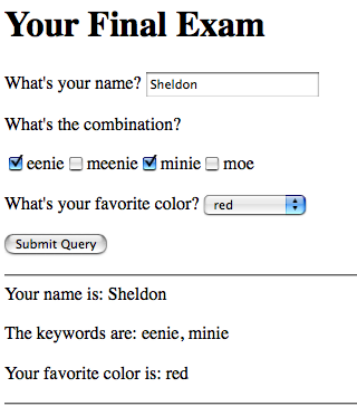
Setting Single- and Multivalued Fields:

```
param(-name=>'tomato',  -value=>'red');
param(-name=>'tomatoes',-value=>['red','green','blue']);
```

Parameters set in this way will be used as default values for fill-out form fields and hidden fields.

A Simple Form

```
#!/usr/bin/perl
# file: final_exam.pl

use CGI ':standard';

print header;
print start_html('Your Final Exam'),
   h1('Your Final Exam'),
   start_form,
   "What's your name? ",textfield(-name=>'first_name'),
   p,
   "What's the combination?",
   p,
   checkbox_group(-name   => 'words',
                  -values  => ['eenie','meenie','minie','moe'],
                  -defaults => ['eenie','minie']),
   p,
   "What's your favorite color? ",
   popup_menu(-name   => 'color',
              -values => ['red','green','blue','chartreuse']),
   p,
   submit,
   end_form,
   hr;

if (param()) {
   print
      "Your name is: ",param('first_name'),
      p,
      "The keywords are: ",join(", ",param('words')),
      p,
      "Your favorite color is: ",param('color'),
      hr;
}

print end_html;
```



Form Generating Functions I

Run *perldoc CGI* for details:

start_form
: Start the form (returns the <form> tag with default attributes).

end_form
: End the form by returning the </form> tag.

textfield(-name=>$name,-value=>$starting_value)
: Create a text field.

password_field(-name=>$name,-value=>$starting_value)
: Create a password field.

textarea(-name=>$name,-value=>$starting_value,-rows=>$rows,-cols=>$cols)
: Create a multiline text input area.

checkbox(-name=>$name,-value=>$value,-checked=>1)
: Create a single checkbox.

checkbox_group(-name=>$name,-value=>\@values,-default=>\@on)
: Create a group of checkboxes sharing the same name. @values gives the list of checkbox values, and @on gives the list of those that are initially on.

Form Generating Functions II

radio_group(-name=>$name,-value=>\@values,-default=>$on)
Create a group of radio buttons sharing the same name. @values gives the list of radio values, and $on indicates which one is on to start with.

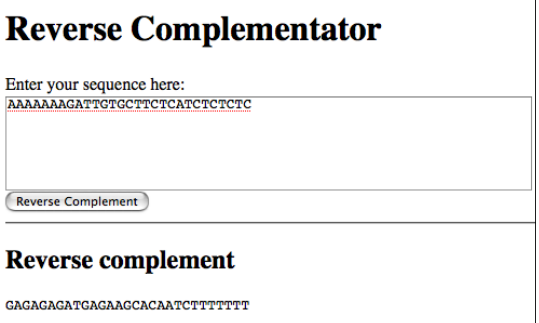
popup_menu(-name=>$name,-value=>\@values,-default=>$on)
Create a popup menu. @values gives the list of items, and $on indicates which one is initially selected.

scrolling_list(-name=>$name,-value=>\@values,-default=>$on)
Create a scrolling list. @values gives the list of items, and $on indicates which one (if any) is initially selected.

submit(-name=>$name,-value=>$value)
Creates a submit button. $value optionally sets the button label.

```
#!/usr/bin/perl
# file: reversec.pl

use CGI ':standard';

print header;
print start_html('Reverse Complementation'),
  h1('Reverse Complementator'),
  start_form,
  "Enter your sequence here:",br,
  textarea(-name=>'sequence',-rows=>5,-cols=>60),
  submit('Reverse Complement'),
  end_form,
  hr;

if ( param ) {
  my $sequence = param('sequence');

  my $reversec = do_reverse($sequence);
  $reversec    =~ s/(.{60})/$1\n/g;  # do word wrap

  print h2('Reverse complement');
  print pre($reversec);
}

print end_html;

sub do_reverse {
  my $seq = shift;
  $seq =~ s/\s//g;              # strip whitespace
  $seq =~ tr/gatcGATC/ctagCTAG/; # complement
  $seq = reverse $seq;          # and reverse
  return $seq;
}
```

A reverse complementation script



File Uploading

HTML: <INPUT TYPE="FILE"> CGI.pm: filefield()

Annoying complication:
You have to start the form with start_multipart_form() rather than start_form().

Let's modify reversec.pl to support file uploads:

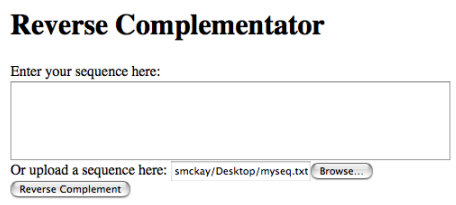
• First part (script too big for one page), print the form

```
#!/usr/bin/perl
# file: sequpload.pl

use CGI ':standard';

print header;
print start_html('Reverse Complementation'),
  h1('Reverse Complementator'),
  start_multipart_form,
  "Enter your sequence here:",br,
  textarea(-name=>'sequence',-rows=>5,-cols=>60),br,
  'Or upload a sequence here: ',filefield(-name=>'uploaded_sequence'),
  submit('Reverse Complement'),
  end_form,
  hr;
```

sequpload.pl continued…

```
if ( param ) {

  my $sequence;

  # look for the uploaded sequence first...
  if ( my $upload = param('uploaded_sequence') ) {
    print h2("Reverse complement of $upload");

    while (my $line = <$upload>) {
      chomp $line;
      next unless $line =~ /^[gatcnGATCN]/;
      $sequence .= $line;
    }

  } else { # ... not found, so read it from the text field
    print h2('Reverse complement');
    $sequence = param('sequence');
  }

  $reversec = do_reverse($sequence);
  $reversec    =~ s/(.{60})/$1\n/g;  # do word wrap
  print pre($reversec);
}

print end_html;

sub do_reverse {
  my $seq = shift;
  $seq =~ s/\s//g;              # strip whitespace
  $seq =~ tr/gatcGATC/ctagCTAG/; # complement
  $seq = reverse $seq;          # and reverse
  return $seq;
}
```

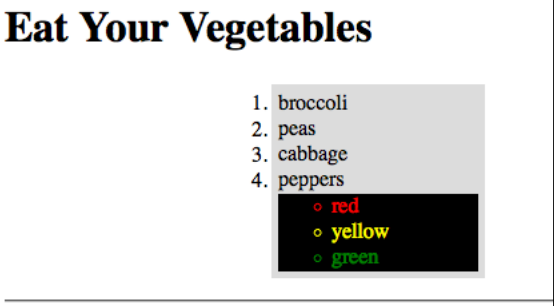
If param() returns true, that means that we have some user input



http://mckay.cshl.edu/cgi-bin/course/sequpload.pl

Adding Cascading Stylesheets

```
#!/usr/bin/perl -w
# Script: veggies_with_style.pl
use CGI ':standard';

my $css = <<END;
<style type="text/css">
 li.yellow { color: yellow }
 li.green  { color: green  }
 li.red    { color: red    }
 ol {
   background-color: gainsboro;
   padding: 5px;
   margin-left: 200px;
   width: 150px;
 }
 ul { background-color: black }
</style>
END

print header,
    start_html( -title => 'Vegetables',
                -head  => $css );
print
    h1('Eat Your Vegetables'),
    ol(
       li(['broccoli', 'peas', 'cabbage']),
       li('peppers',
          ul(
             li({-class => 'red'},'red'),
             li({-class => 'yellow'},'yellow'),
             li({-class => 'green'},'green')
             )
          ),
       ),
    hr,
    end_html;
```



http://mckay.cshl.edu/cgi-bin/course/veggies_with_style.pl

External stylesheet

```
#!/usr/bin/perl -w
# Script: veggies_with_style.pl
use CGI ':standard';

my $css = '/css/veggies.css';

print header,
    start_html( -title => 'Vegetables',
                -style  => $css );
print
    h1('Eat Your Vegetables'),
    ol(
       li(['broccoli', 'peas', 'cabbage']),
       li('peppers',
          ul(
             li({-class => 'red'},'red'),
             li({-class => 'yellow'},'yellow'),
             li({-class => 'green'},'green')
             )
          ),
       ),
    hr,
    end_html;
```

http://mckay.cshl.edu/cgi-bin/course/veggies_with_style2.pl

CGI Exercises
Problem #1

Write a CGI script that prompts the user for his or her name and age. When the user presses the submit button, convert the age into "dog years" (divide by 7) and print the result.

Problem #2

Accept a DNA sequence and break it into codons.

Extra credit: Translate the codons into protein.

Overview and Applications of Next-Generation Sequencing Technologies

Stéphane Deschamps

Analytical & Genomic Technologies

DuPont Agricultural Biotechnology

Outline

1. Next-Generation Sequencing Platforms

2. Applications of Next-Generation Sequencing Technologies
    1. Overview
    2. Variant detection with Illumina platform

3. Third-Generation Sequencing technologies: what's next?

Sanger sequencing

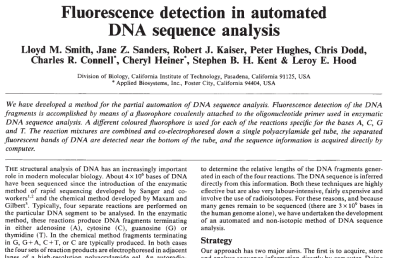
DNA sequencing with chain-terminating inhibitors

(DNA polymerase/nucleotide sequences/bacteriophage øX174)

F. SANGER, S. NICKLEN, AND A. R. COULSON

Medical Research Council Laboratory of Molecular Biology, Cambridge CB2 2QH, England

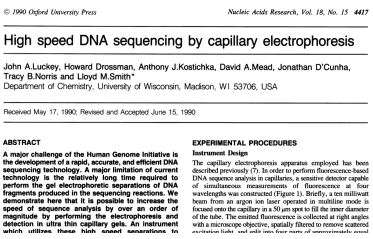
Contributed by F. Sanger, October 3, 1977



Fluorescence detection in automated DNA sequence analysis

Lloyd M. Smith, Jane Z. Sanders, Robert J. Kaiser, Peter Hughes, Chris Dodd, Charles R. Connell*, Cheryl Heiner*, Stephen B. H. Kent & Leroy E. Hood

Division of Biology, California Institute of Technology, Pasadena, California 91125, USA
* Applied Biosystems, Inc., Foster City, California 94404, USA

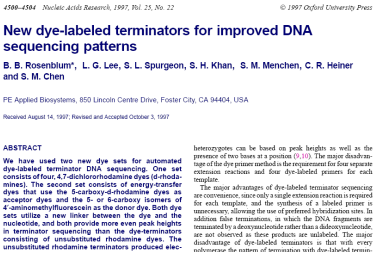


© 1990 Oxford University Press — Nucleic Acids Research, Vol. 18, No. 15 4417

High speed DNA sequencing by capillary electrophoresis

John A.Luckey, Howard Drossman, Anthony J.Kostichka, David A.Mead, Jonathan D'Cunha, Tracy B.Norris and Lloyd M.Smith*
Department of Chemistry, University of Wisconsin, Madison, WI 53706, USA

Received May 17, 1990; Revised and Accepted June 15, 1990



4500–4504 Nucleic Acids Research, 1997, Vol. 25, No. 22 — © 1997 Oxford University Press

New dye-labeled terminators for improved DNA sequencing patterns

B. B. Rosenblum*, L. G. Lee, S. L. Spurgeon, S. H. Khan, S. M. Menchen, C. R. Heiner and S. M. Chen

PE Applied Biosystems, 850 Lincoln Centre Drive, Foster City, CA 94404, USA

Received August 14, 1997; Revised and Accepted October 3, 1997

Successive improvements now allows 96 800-900 base reads to be sequenced in less than 2h

Sanger sequencing

Sanger sequencing has been, and still is, very useful...

...but it remains slow and expensive

Sequencing Platform Comparisons

	ABI 3730xl	454 FLX Titanium	Illumina HiSeq 2000
Read Length	700-800bps	500bps	>100bps
Number of Reads/Run	96	1MM	2,000MM
Max Yield/Run	~70Kbps	0.5Gbps	200Gbps
Cost/1Gbp	$3.5MM	$15,000	$125
Run time/Instrument to 1Gbp	8 years	1 day	1 hour

Next-Generation Sequencing

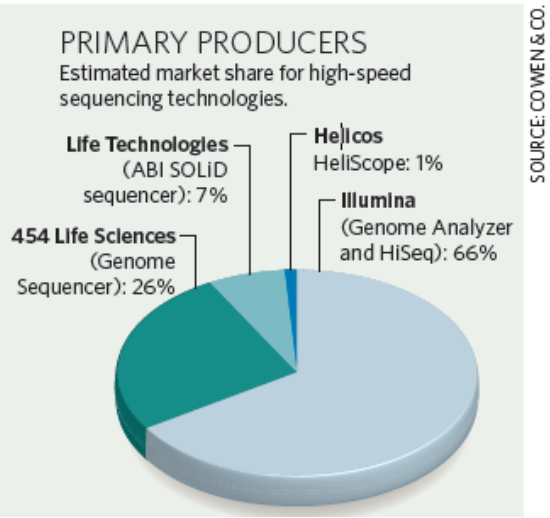
Second-generation platforms:

- 454/Roche
- HiSeq/Illumina
- SOLiD/Life Technologies
- Heliscope/Helicos BioSciences

Third-generation platforms:

- Ion Torrent
- SMS/Life Technologies
- Pacific Biosciences
- Intelligent Bio-Systems
- ZS Genetics
- LightSpeed Genomics
- NABsys
- Oxford Nanopore Technologies
- ...

Next-Generation Sequencing



Nature, 467, September 2010

454 FLX Titanium

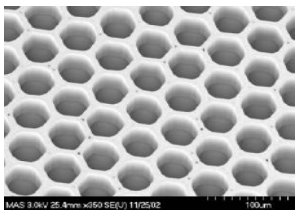
- First next-generation sequencing platform launched (October 2005)
- Titanium chemistry for the 454 FLX launched in September 2008
- Sequencing By Synthesis
  - Pyrosequencing
  - Chemiluminescent signal
- Long read technology (~500 nucleotides)
- Generates up to 0.5Gbps per run

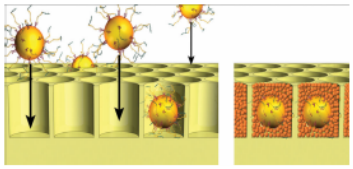
454 FLX Titanium

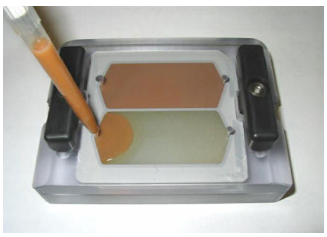
- DNA Library Construction
  - Ligation of A & B adaptors
  - No cloning
- Emulsion PCR
  - DNA capture beads
  - Clonal amplification in “microreactors”
- Pyrosequencing
  - Deposition of enriched beads into picotiter plate

Bead deposition into plates

- Deposition of enriched beads into PicoTiter plate
- Well diameter = 29uM allowing for a single bead (20uM diameter) per well
- Chambers are filled with enzyme beads, DNA beads and packing beads.



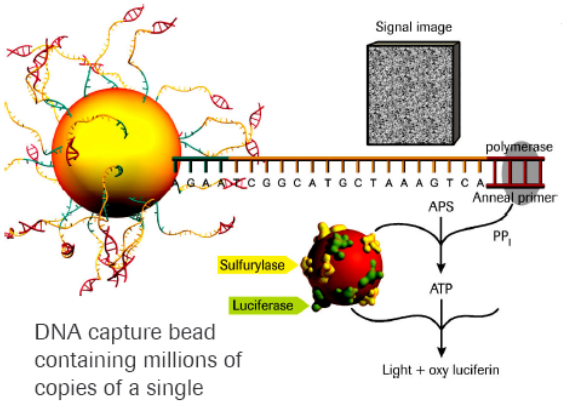




www.roche-applied-science.com

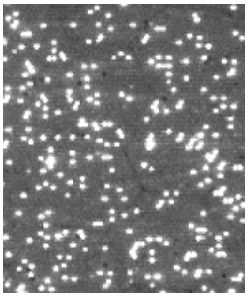
Pyrosequencing

1. Polymerase add nucleotide (sequential flow of dNTPs)
2. PPi is released
3. Sulfurylase creates ATP from PPi
4. Luciferase hydrolyzes ATP and use luciferin to make light

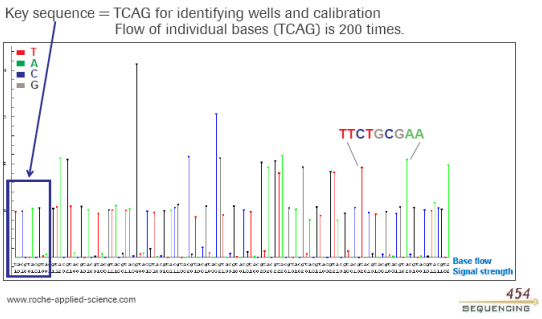


www.roche-applied-science.com

Image and signal processing



1. Raw data is series of images (one image per base per cycle).
2. Data are extracted, quantified and normalized.
3. Read data are converted into “flowgrams”.



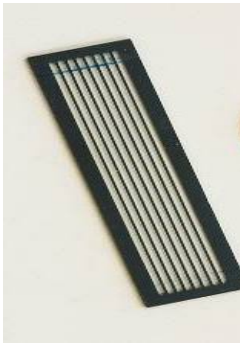
Post-processing

1. Output = flowgrams, basecalls, Phred-equivalent scores

2. Basecall & Flowgrams can be used in the following applications:
   1. **De novo assembler** – consensus sequences assembled into contigs with quality scores and ACE file (works best with genomic DNA).
   2. **Reference mapper** – contigs mapped to reference sequence + list of high-confidence mutations
   3. **Amplicon variant analyzer** – identification of sequence variants in amplicon libraries

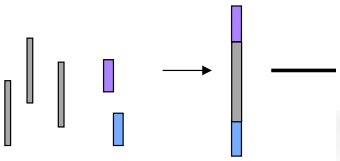
Illumina HiSeq 2000

- Single molecule array ("flow cell") with tens of millions of amplified clusters
- Sequencing By Synthesis
  - Removable fluorescence
  - Reversible terminators
- Short read technology (>150 nucleotides)
- Generates >200Gbps per run



Illumina HiSeq 2000

Sample Prep | **cBot** | **HiSeq 2000**



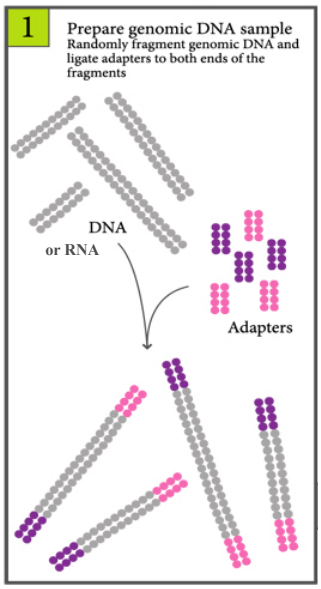
Prepare DNA fragments + Ligate adapters

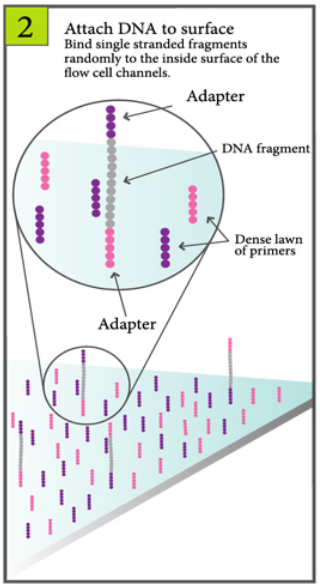
Cluster Synthesis

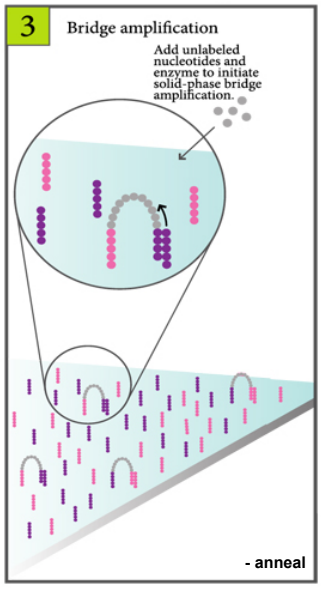
Sequencing

Analysis Pipeline

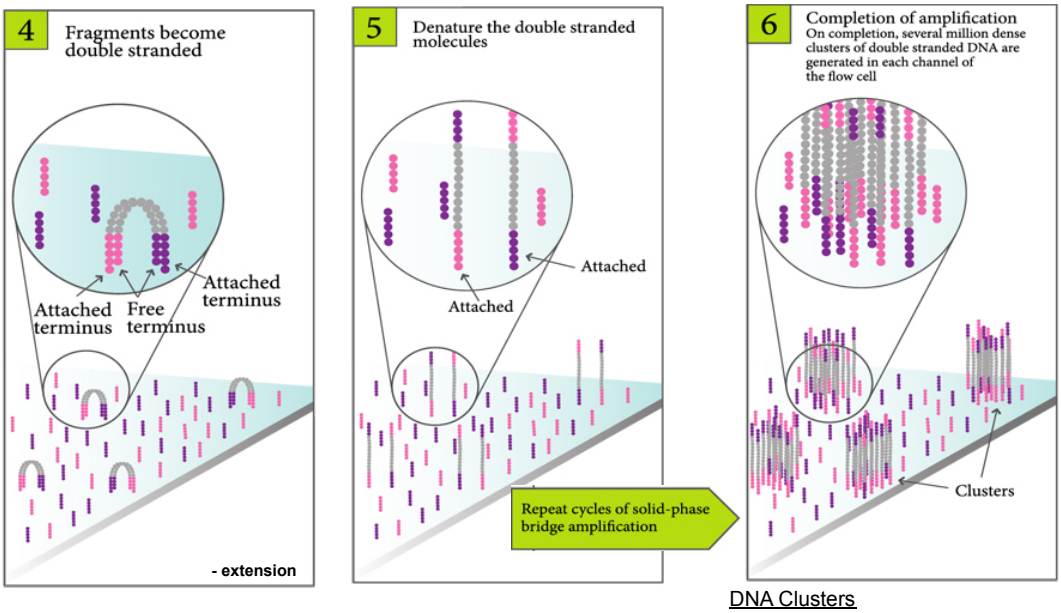
Cluster Generation







Cluster Generation

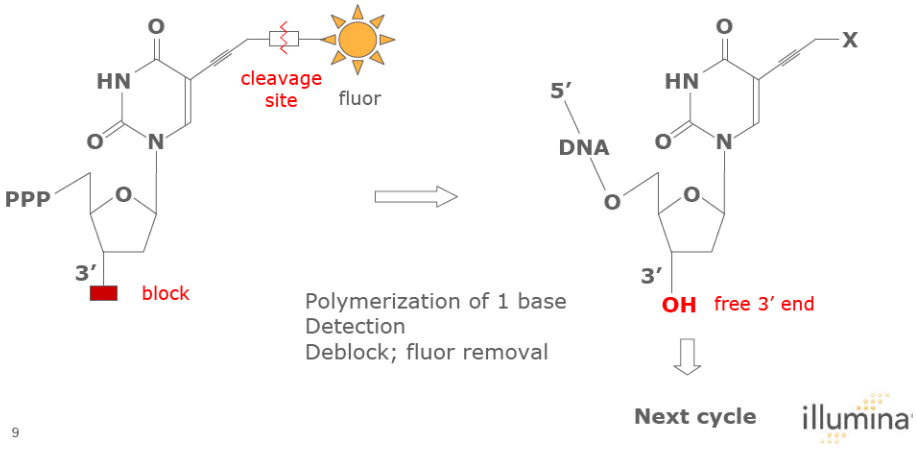


DNA Clusters

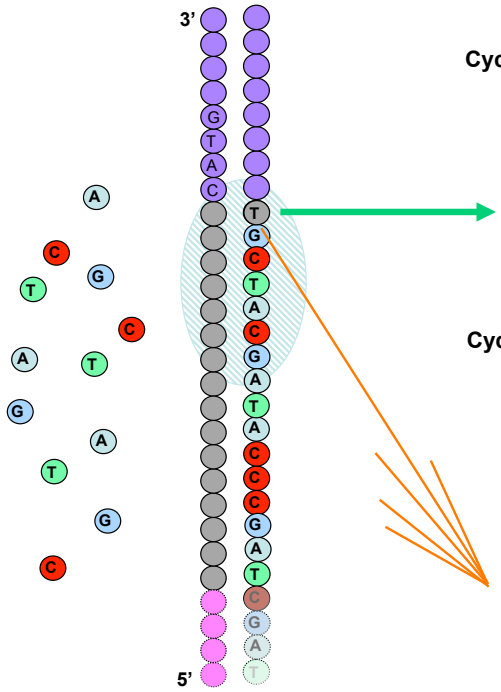
- ~1,000 copies of DNA in each cluster
- 1-2 microns in diameter

Reversible Terminator Chemistry

- Incorporation
- Imaging
- Cleavage



Sequencing by Synthesis (SBS)



Cycle 1: Add sequencing reagents

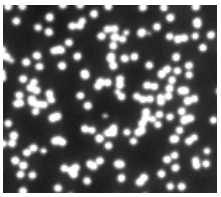
First base incorporated

Remove unincorporated bases

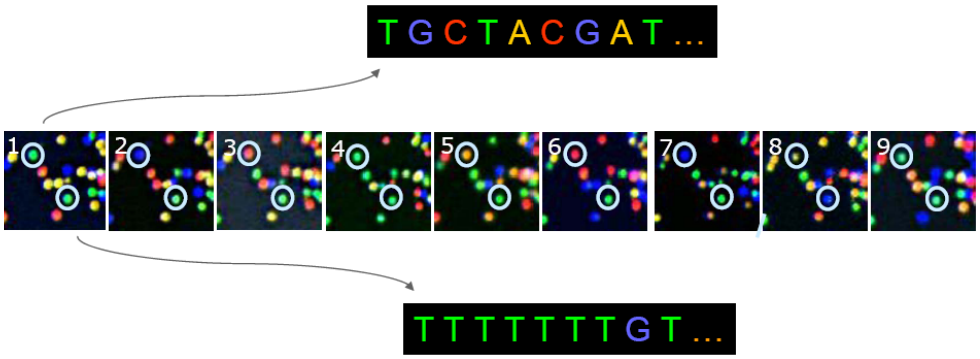
Detect signal

Deblock (removal of fluorescent dye and protecting group)

Cycle 2 - n: Add sequencing reagents and repeat



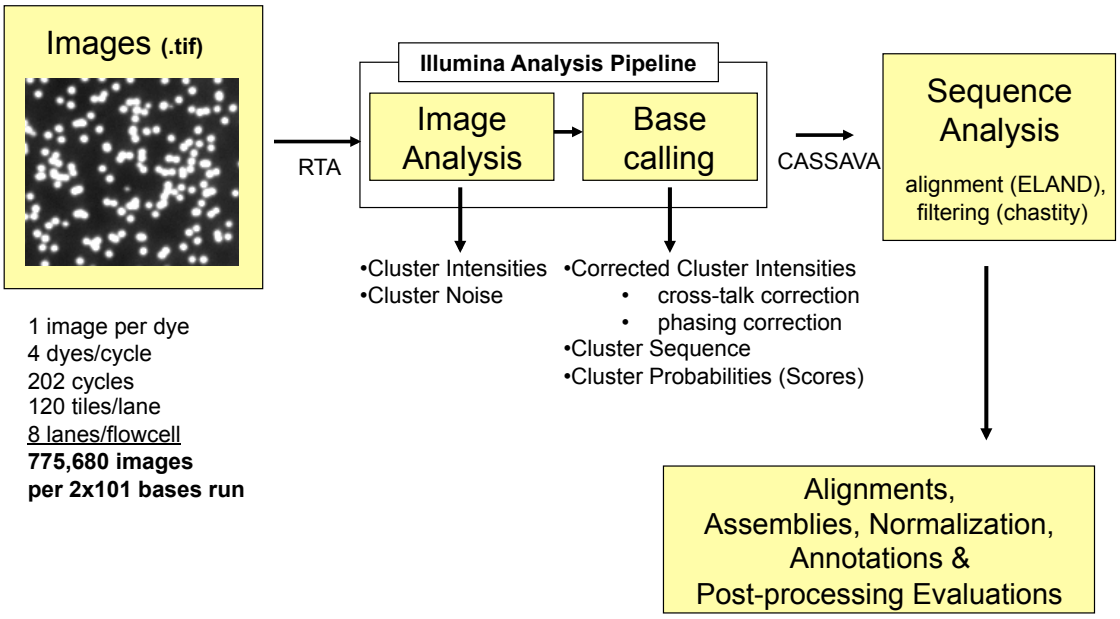
Sequencing by Synthesis (SBS)



The identity of each base of a cluster is read off from sequential images

illumina

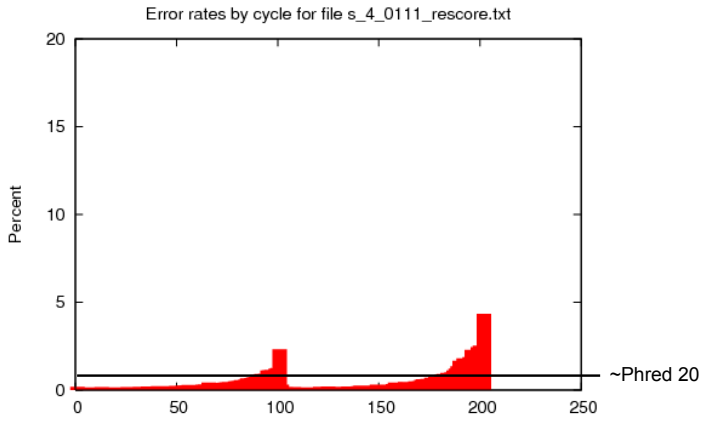
Data Analysis Workflow - Illumina



Other platforms

Sequencing Platform	Sequencing Chemistry	Run Time	Read Length (bp)	Reads per Run (million)	Throughput per Run (Gbp)
Roche 454 FLX	Pyrosequencing	10h	400-500	~1	0.4-0.5
Illumina HiSeq	Sequencing by Synthesis	8 days	100	>2,000	200
ABI SOLiD 4	Sequencing by Ligation	12-16 days	50	>1,400	80-100 (mappable)
Helicos HeliScope	Sequencing by Synthesis	8 days	25-55	600-1,000	21-35
Polonator	Sequencing by Ligation	80h	28	300-400	10

Data Quality

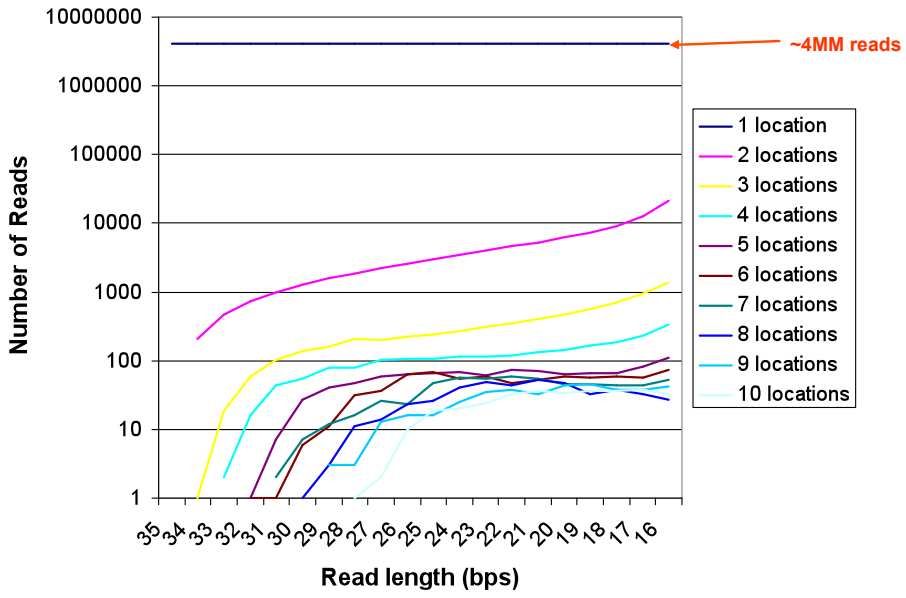


Phred score 20 = 1% error rate

Quality *vs.* Read Length? Trimming?

Single short read uniqueness

Illumina 35 base reads aligned to A. thaliana genome



Applications of Next-Generation Sequencing

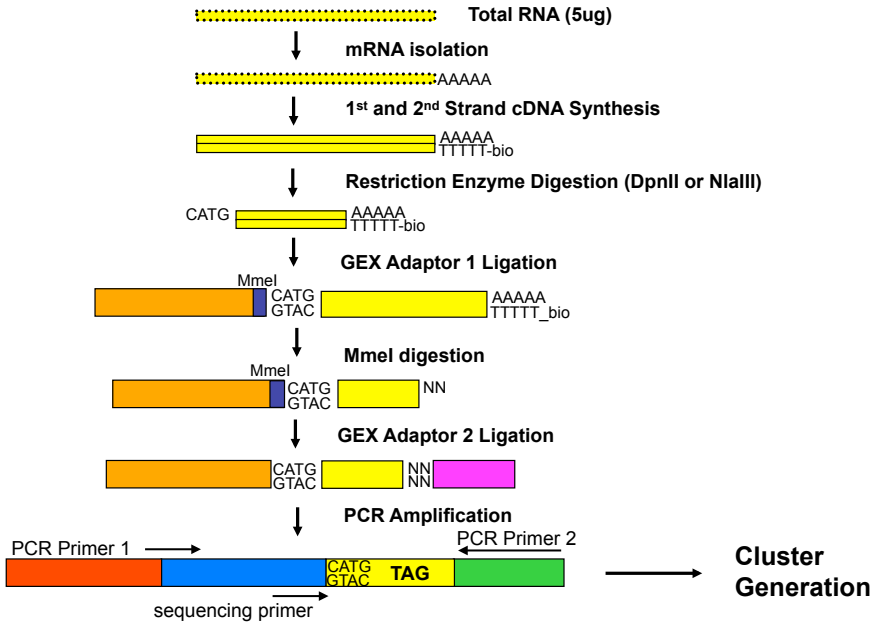
Gene Expression Profiling

- **Tag count & Alignments**

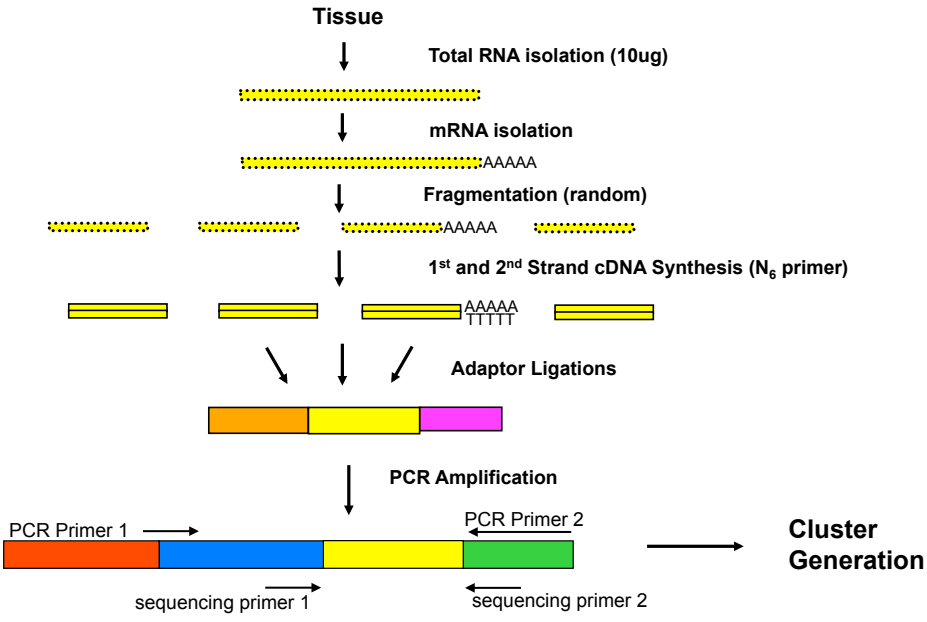
- **Digital Gene Expression Tag Profiling**
  - Short cDNA fragments mapping to 3' ends of transcripts
  - SAGE-like approach (1 short tag/transcript)
  - 20 base tag output (RE site + 16 bases) aligned to a reference genome
  - Identify, quantify and annotate expressed genes

- **Transcriptome Profiling (RNA-Seq)**
  - cDNA fragments generated via random priming
  - 100 base output aligned to a reference genome
  - Assemble entire transcript sequence
  - Identify, quantify and annotate expressed genes
  - Identify SNPs, alleles and alternative splice variants

Tag Profiling – Sample Prep (Illumina)



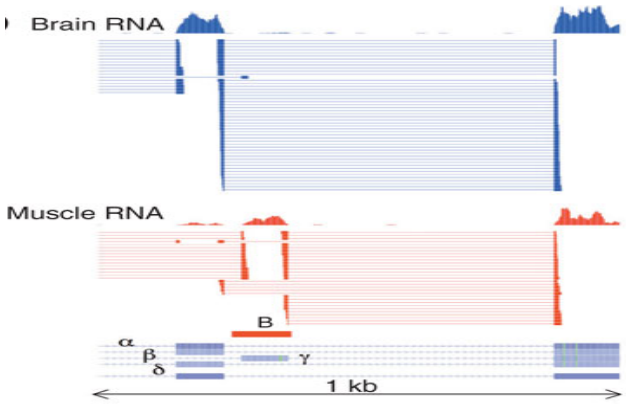
Transcriptome Profiling – RNA-Seq (Illumina)



Novel Transcript Discovery

– **Small RNA Identification and Profiling**
  - Small RNA size is suitable to discovery with next-generation sequencing

– **Deep assessment of alternative splicing isoforms**
  - Deep coverage allows discovery of rare isoforms



Mortazavi et al. (2008), Nat. Methods

De novo Sequencing

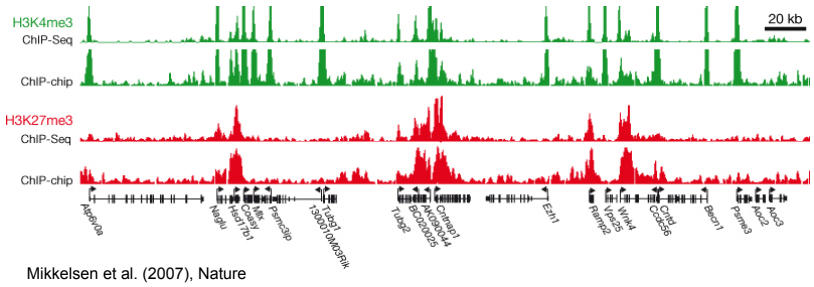
– **Whole Genome & Transcriptome Sequencing**
  - Small genomes that are not too complex (microbial)
  - The longer the reads, the better – 454 chemistry most suitable
  - Recent improvements of Illumina platform - ~500Kbps contigs
  - Paired-End sequencing

– **Targeted Sequencing**
  - Indexed PCR products
    - Raindance Technologies
    - Padlock probes
  - Indexed BAC clones
  - Sequence Capture (Solid phase, Liquid phase)
    - Agilent, Febit & Nimblegen
  - Reduced-Representation (Methyl-sensitive Enzymes)

Gene Regulation

- **ChIP-Seq** (immunoprecipitate sequencing)
  - Capture regions of the genome bound by proteins (transcription factors, histones)
  - Requires complex algorithm to determine differential levels of coverage throughout the genome

- **Methyl-Seq** (methylation status) – Bisulfite Sequencing
  - Technically complex
  - Requires alignment of distinct modified DNA strands to reference sequence



Mikkelsen et al. (2007), Nature

Methyl-Seq

RBS_MAIZE Ribulose bisphosphate carboxylase small chain (pco504677)

ACAAGGAGCTGCAGGAGGCCATCAAATCCTACCCGGACGCCTTCCACCGCGTCATCGGCTTCGACAACATCAAGCAGACGCAG
TGCGTCAGCTTCATCGCCTACAAGCCCCCGGGCAGCGACTAGACCGCGCCCGCCGGCCGCCCCCCGCCGGCTAGCTAGCTAGC
TAGCTCCTGCGTGAGCTAGTAGCTAGTGCCATGCGTCGTCTCTGTCGTTCGGTTTTGCTTCGGGTCACCGTGTACCCTTTGCTTG
CTTGGTTTCTTCTTTCCTTTTTTCCTTTTTTTTTCTTCTTTTCCCCGGCCATGGTTCCTTTGCTTTCCAGCAGTTCTCTGCTGGATGT
GATGTATCCATTGTTGCAAGCATGCATGGCCTTGCATTGGCTACCTCTATACCTGCTACAAAACTACTGCAACGCCTATATATAC
TTGGGGTGAGGAACATGTGAATGCAAGCTCCGGCTATCATATACATGTAATATGGATACAAACTATATATATAAATCCGCCGA
GGCGCCGACAATACTATACGACGACACCGTGTTAAGTTAATATATAACTGGTGCTTTTTATTCAAAAGTCGACG

ORIGINAL TARGET
C+G content = 50%
Identity to Reference = 100%

ATAAGGAGTTGTAGGAGGTTATTAAATTTTATTTGGATGTTTTTTATTGTGTTATTGGTTTTGATAATATTAAGTAGATGTAGTG
TGTTAGTTTTATTGTTTATAAGTTTTTGGGTAGTGATTAGATTGTGTTTGTTGGTTGTTTTTTGTTGGTTAGTTAGTTAGTTAGTT
TTTGTGTGAGTTAGTAGTTAGTGTTATGTGTTGTTTTTGTTGTTTGGTTTTGTTTTGGGTTATTGTGTATTTTTTGTTTGTTTGGT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTGGTTATGGTTTTTTGTTTTTTAGTAGTTTTTTGTTGGATGTGATGTATT
TATTGTTGTAAGTATGTATGGTTTTGTATTGGTTATTTTTATATTTGTTATAAAATTATTGTAATGTTTATATATATTTGGGGTGA
GGAATATGTGAATGTAAGTTTTGGTTATTATATATATGTAATATGGATATAAATTATATATATAAATTTGTTGAGGTGTTGATA
ATATTATATGATGATATTGTGTTAAGTTAATATATAATTGGTGTTTTTTATTTAAAAGTTGATG

Sequence 1 (from Sense)
C+G content = 22%
Identity to Reference = 71%

Sense/antisense Identity ~ 30%

ACAAAAAACTACAAAAAACCATCAAATCCTACCCAAACACCTTCCACCACATCATCAACTTCAACAACATCAAACAAACACA
ATACATCAACTTCATCACCTACAAACCCCCAAACAACACAACTAAACCACACCCACCAACCACCCCCCACCAACTAACTAACTAA
CTAACTCCTACATAAACTAATAACTAATACCATACATCATCTCTATCATTCAATTTTACTTCAAATCACCATATACCCTTTACTT
ACTTAATTTCTTCTTTCCTTTTTTCCTTTTTTTTTTCTTCTTTTCCCCAACCATAATTCCTTTACTTTCCAACAATTCTCTACTAAATA
TAATATATCCATTATTACAAACATACATAACCTTACATTAACTACCTCTATACCTACTACAAAACTACTACAACACCTATATAT
ACTTAAAATAAAAAACATATAAATACAAACTCCAACTATCATATACATATAATATAATACAAACTATATATATAAATCCACC
AAAACACCAACAATACTATACAACACACCATATTAAATTAATATATAACTAATACTTTTTATTCAAAAATCAACA

Sequence 2 (From Antisense)
C+G content = 29%
Identity to Reference = 79%

(slide courtesy of V. Llaca)

Variant & Structural Variation

- **Whole Genome Resequencing**
  - Small genomes that are not too complex (repeats, duplications...)
  - The longer the reads, the better

- **Targeted Resequencing**
  - Complex genomes (crops)
    - Reduced representation libraries (methyl-sensitive enzymes)
    - Transcriptome
  - Sequence Capture (Solid Phase, Liquid Phase)
    - » Agilent, Febit & Nimblegen

- **CNVs** (Copy Number Variants) **& Indels**
  - Paired-end sequencing and alignment to reference sequence

Challenges in variant discovery

1. **Base quality & filtering (scoring threshold)**

2. **Sequencing errors *vs.* SNPs**
   1. To differentiate true polymorphisms from sequencing errors
   2. Coverage of a given SNP region and redundancy of reads (coverage *vs.* number of samples)

3. **Availability of a reference sequence (genome)**
   1. To separate unique *vs.* duplicated sequences
   2. Duplication in one line but not another (CNVs)
   3. Unmappable data (Indels)
   4. Polymorphism rate in one line *vs.* another = need to set conditions for alignment

4. **Paired-end sequencing can help unique read placement**

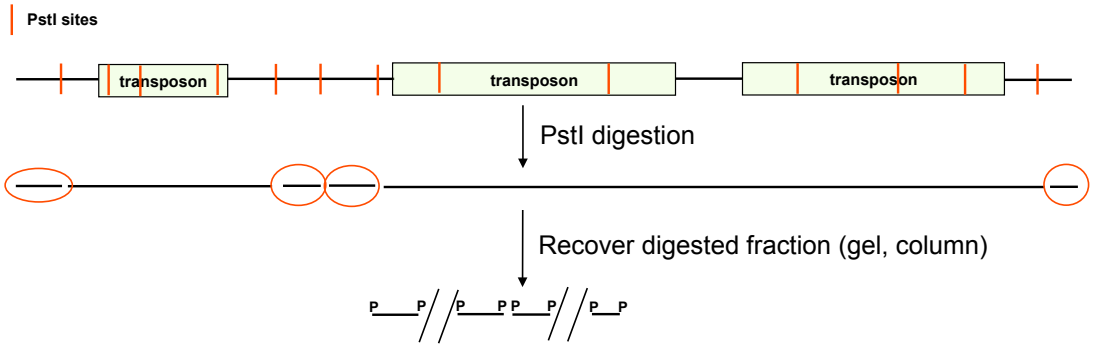
5. **Complex genomes = need to reduce complexity prior to sequencing**
   1. High repeat content (ex: ~80% in maize, ~70% in soy, 90% in sunflower…)
   2. Gene duplications and genome plasticity (polyploidy, partial or whole genome duplications...)

Methylation in plants

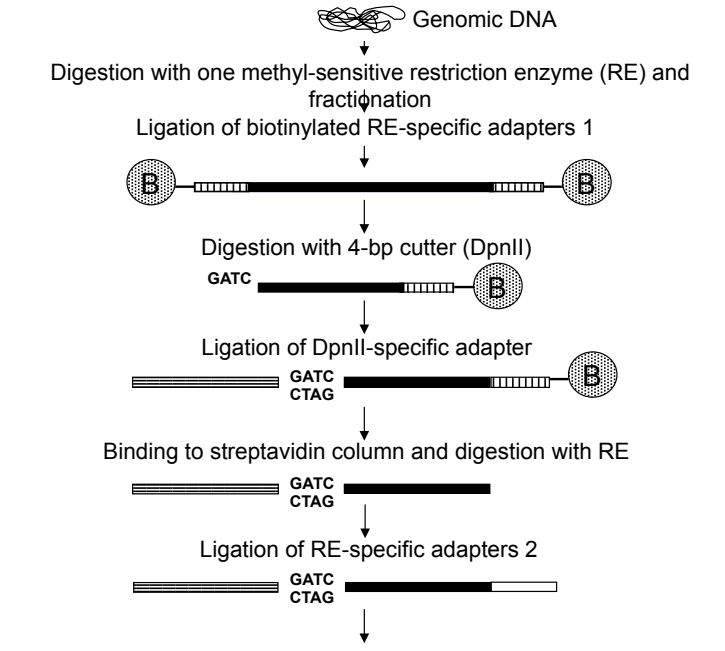
1. DNA methylation in plants occurs at 5-methyl cytosine within CpG dinucleotides and CpNpG trinucleotides

2. Transposons and other repeats comprise the largest fraction of methylated DNA. Studies in Arabidopsis have shown that CG sites in the 3' end of the transcribed regions of more than one third of all genes also are methylated (Zhang, Science, 320, 489, 2008).

3. Methylation is critically important in silencing transposons and regulating plant development (methylation in promoters appears to reduce transcription)



Library Construction



SNP detection flowchart

Filtering and Condensing

Basecalling, cropping last 4 bases & initial base-quality filter (for individual tags)

↓

Condensing & optional consensus base-quality filter (for unitags sequences)

↓

Creating HQ unitag datasets (removing singlets)

Comparing two genotypes

Comparing HQ unitag datasets from genotype "A" and genotype "B" using Vmatch

↓

Filtering, to accept clusters with only two members (A, B) with exactly one mismatch

↓

Recovering matched HQ unitag sequences and SNP sites from Vmatch alignments

Mapping to genome

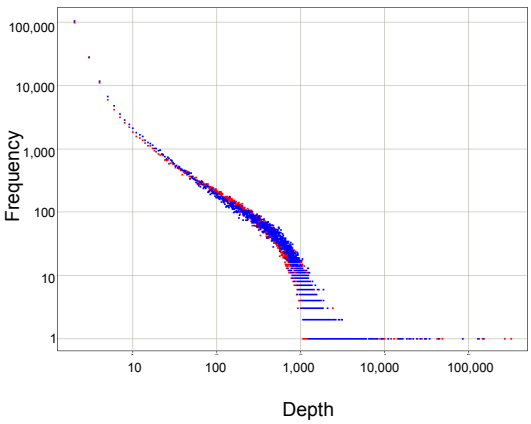
Mapping SNP-containing HQ unitags to reference sequence (genome), using a *k*-mer table (*k*=length of trimmed tags), and find copy numbers and locations.

↓

Capturing single-copy HQ unitags with up to a single-base mismatch to the reference sequence at the exact location of the putative SNP site for one or both genotypes

Example: one flow cell in soybean (Williams82 vs. Pintado)

Run Metrics	Williams82	Pintado
Number of total reads generated (after initial basecalling)	37,666,279	38,000,474
Number of filtered total reads †	24,519,484	23,101,973
Number of unitags (generated from filtered total reads)	965,610	885,429
Number of high quality (HQ) unitags ‡	255,918	246,102
Alignment of HQ unitags against the reference sequence:		
Zero mismatch §	208,923	197,015
One mismatch §	27,770	27,699
Two or more mismatches §	19,225	21,388
HQ unitags aligning uniquely to the reference sequence with zero mismatch	152,185	144,559



† Filtered total reads defined as having a quality value for individual base greater than or equal to 15

‡ HQ unitag reads defined as having a quality value for each base greater than or equal to 15, and with an individual read count greater than or equal to 2.

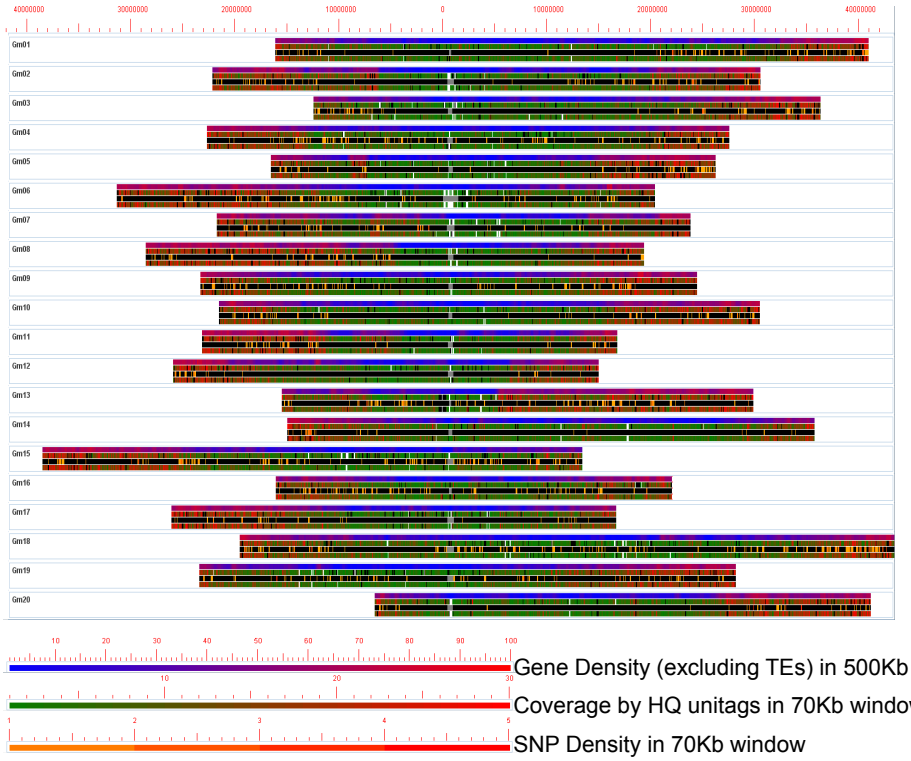
§ Best match to reference sequence of HQ unitag reads aligning uniquely or multiple times to the reference sequence

Results & Validation

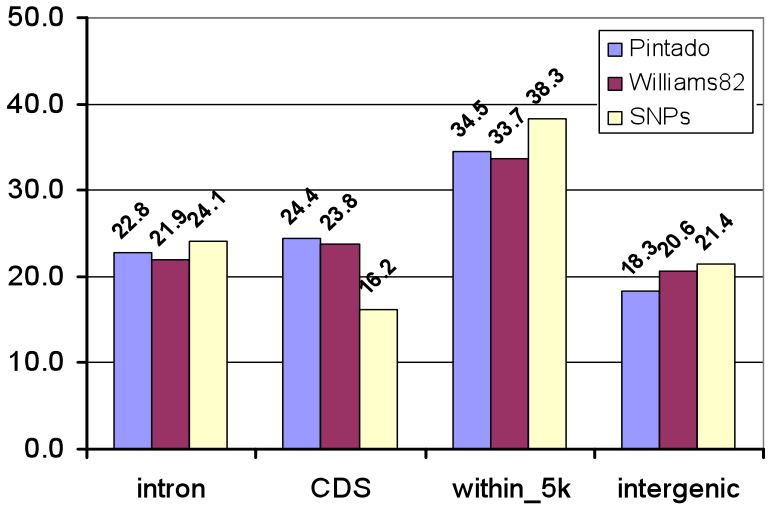
Experiments	Putative SNPs	Confirmed *	Not Confirmed*	Validation rate
Q Score threshold: 15				
Soy: Williams82 vs. Pintado	1,682	163	5	97.0%
Rice: Kasalath vs. Taichung65	2,618	162	6	96.4%
Q Score threshold: 25				
Soy: Williams82 vs. Pintado	702	168	2	98.8%
Rice: Kasalath vs. Taichung65	2,148	174	1	99.4%

*SNPs confirmed/not confirmed via Sanger sequencing of PCR products for both genotypes

Distribution of HQ unitags & SNPs related to annotated gene density (soybean)



Distribution of HQ unitags & SNPs related to distance to annotated genes (excluding TEs) in soybean



Intron, CDS and UTR coordinates determined from GFF annotation files

Bioinformatic tools

Alignment and Polymorphism Detection

1. SOAP – Short Oligonucleotide Alignment Program
   - Ruiqiang Li, Beijing Genomics Institute
   - http://soap.genomics.org.cn
2. MAQ – Mapping and Assembly with Quality
   - Heng Li, Sanger Centre
   - http://maq.sourceforge.net/maq-man.shtml
3. Bowtie - An ultrafast memory-efficient short read aligner
   - Ben Langmead and Cole Trapnell, University of Maryland
   - http://bowtie-bio.sourceforge.net/

Bioinformatic tools

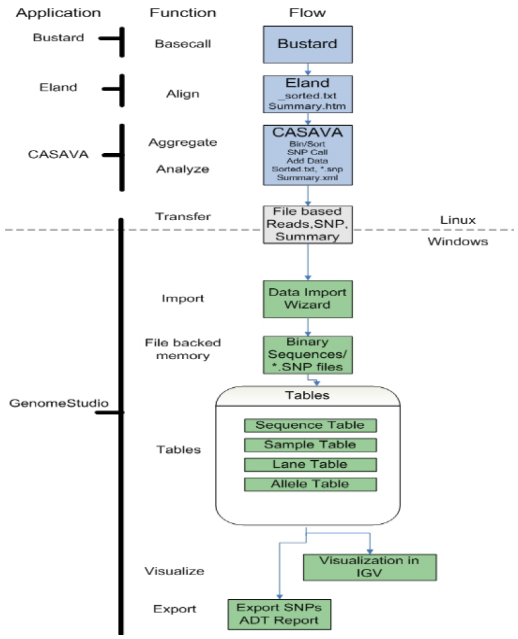
Genomic Assembly

1. Velvet – De novo assembly of short reads
   - Daniel Zerbino and Ewan Birney, EMBL-EBI
   - http://www.ebi.ac.uk/~zerbino/velvet/
2. SSAKE – Assembly of short reads
   - Rene Warren, et al, British Columbia Cancer Agency
   - http://bioinformatics.oxfordjournals.org/cgi/content/full/23/4/500
3. Euler – Genomic Assembly
   - Pavel Pevzner and Mark Chaisson, University of California, San Diego
   - http://nbcr.sdsc.edu/euler/

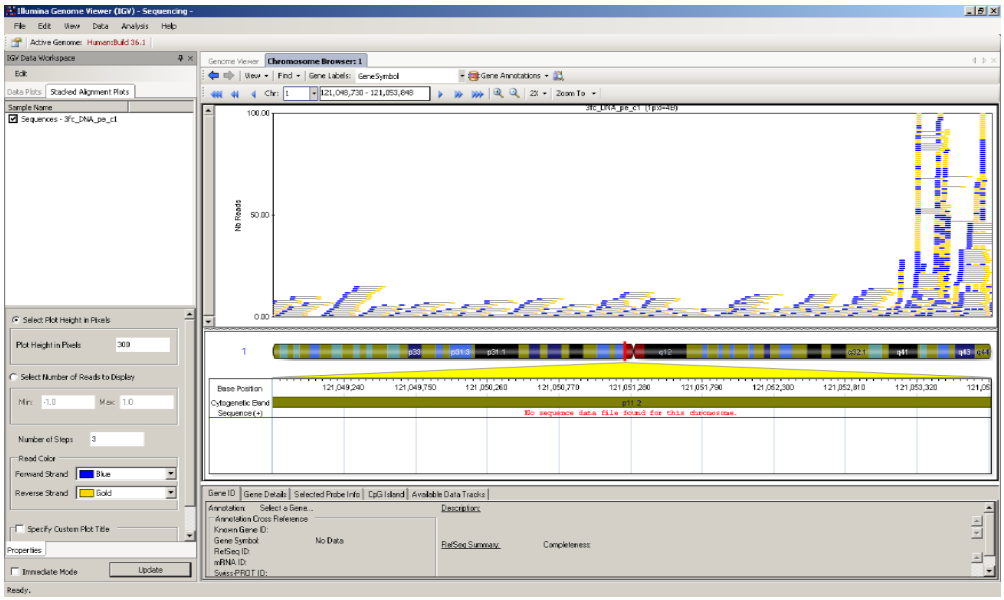
Bioinformatic tools - Illumina

Overview

1. Obtain Bustard reads and align against Genome with Eland
2. Aggregate and SNP call data with CASAVA
3. GenomeStudio™ wizard import of data
4. Examine coverage and quality in stacked alignment graphs for a selected region/chromosome
5. Export table of SNPs and consensus sequence



Bioinformatic tools - Illumina



Third-Generation Sequencing technologies: what's next?

Next-Generation Sequencing

Second-generation platforms:

- *454/Roche*
- *HiSeq/Illumina*
- *SOLiD/Life Technologies*
- *Heliscope/Helicos BioSciences*

Third-generation platforms:

- Ion Torrent
- SMS/Life Technologies
- Pacific Biosciences
- Intelligent Bio-Systems
- ZS Genetics
- LightSpeed Genomics
- NABsys
- Oxford Nanopore Technologies
- ...

Next-Generation Sequencing

Known issues related to second-generation sequencing platforms:

1) Amplification bias
    1) Non-uniform amplification of DNA that leads to over-representation of certain sequences and under-representation of others

2) Crosstalk
    1) Overlap between signals for different nucleotides in a sequencing reaction (emission spectrum of two fluorophores may overlap)

3) Dephasing
    1) Sequence reads from ensemble of molecules representing a single input sequences gradually diverge in length in “wash-and-scan” techniques

4) De novo assembly
    1) Limited assembly of large genomes (repeats)

Next-Generation Sequencing

The answer: single-molecule sequencing!

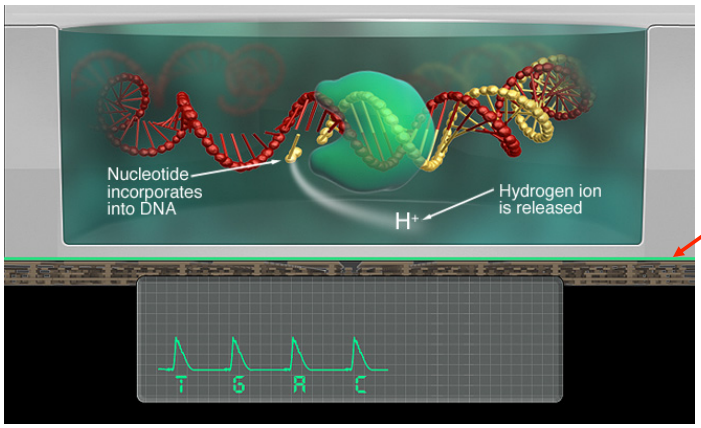
1) Single molecules of DNA observed as they are synthesized by single DNA polymerase (Pacific Biosciences, Life Technologies...)
    1) Problem: missing bases (spectral overlap)

2) Single molecules of DNA threaded through a nanopore or positioned at proximity of a nanopore (Oxford Nanopore, NABsys...)
    1) Problem: speed of detection & parallelization

3) Single molecules of DNA imaged using electronic microscopy techniques (ZS Genetics, Halcyon...)
    1) Problem: upfront costs and highly trained personnel

4) Single molecules of DNA ligated to DNA probes using microfluidic techniques (GnuBio)
    1) Problem: ?

Ion Torrent



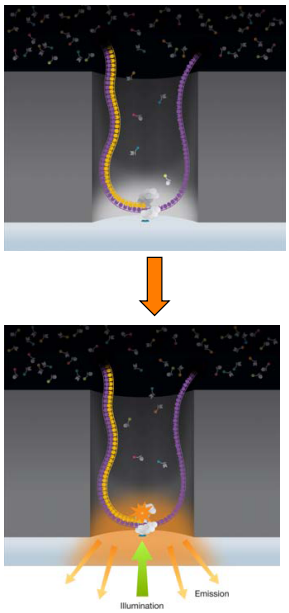
ion sensor:
1. Detect pH change caused by released of H +
2. Generated charge turned into a voltage

http://www.iontorrent.com/the-simplest-sequencing-chemistry/

1. $50K instrument (<50lbs)
2. 1.4MM sub-microscopic wells
3. 100-200 bps / reaction
4. No imaging required: limited need for data storage & management

Pacific Biosciences

- SMRT™ Technology
- Single DNA polymerase attached at bottom surface of nanometer-scale hole (“ZMW”), incorporates in real-time fashion fluorescently labeled nucleotides to elongated strand of DNA
- Elongated strand can be several thousands of nucleotides in length



www.pacificbiosciences.com

Pacific Biosciences

1. Small size of the hole favors rapid in-and-out diffusion of nucleotides and dye following their cleavage. Meanwhile, incorporated nucleotide is held within the detection volume for 10’s of milliseconds, order of magnitude longer than the time it takes for nucleotides to diffuse in and out of the hole, therefore decreasing background noise

2. Fluorescent dye is attached to the phosphate chain, rather than the base, and is cleaved when the nucleotide is incorporated to the DNA strand.

=> Decreased background noise and use of phospholinked nucleotides circumvents the need for successive cycles of incorporation, washing, scanning and removal of the label, therefore optimizing processivity of the enzyme and allowing longer read lengths

Pacific Biosciences

1. Current specs:
    1. 2 x 80,000 ZMWs per SMRT cell (~30% of ZMWs generate data)
    2. Polymerase incorporates 1-3 nucleotides per second
    3. ~1,000-1,250bps average read lengths => ~30Mbps/15 minutes (up to 96 SMRT cell/48 hours)
    4. Strobe sequencing: 4x250bps, 2x500bps...
    5. 99.99% **consensus** accuracy

2. Projected specs (2011-2012):
    1. 4 x 80,000 ZMWs per SMRT cell (~90% of ZMWs generate data)
    2. Polymerase incorporates 10-15 nucleotides per second (~10-20Kbps reads)

3. V2 machine (2014):
    1. No camera: each ZMW assigned its own detector (“optode”)
    2. Several millions optodes per SMRT cells
    3. Polymerase incorporate up to 50 nucleotides per second (~50-100Kbps reads)
    4. >100Mbps/second

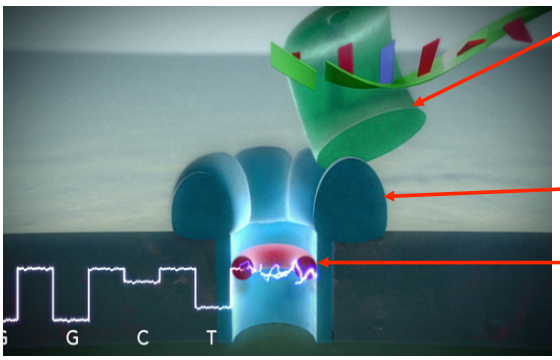
Oxford Nanopore

Protein nanopore

- Long read lengths (1000’s)
- High read accuracy
- Current technical issues:

  1) Attachment of the exonuclease to the pore

  2) Parallelization (1,000’s of pores per chips)



Exonuclease

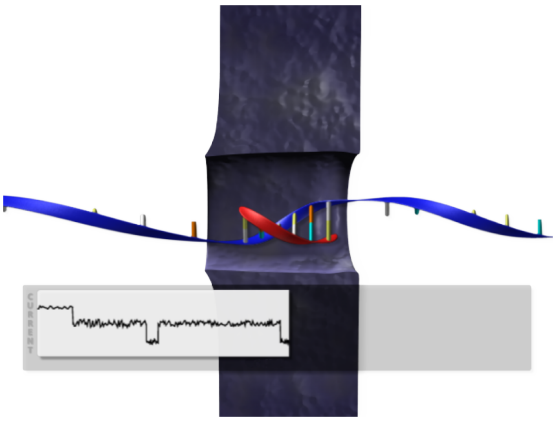
Alpha-hemolysin

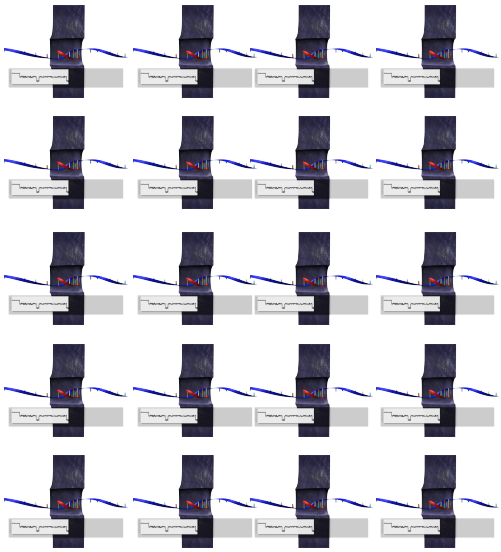
Cyclodextrine (encapsulate nucleotide)

http://www.technologyreview.com/biomedicine/22220/

NABsys

Sequencing-by-Hybridization + Nanopores (Electrical detection)

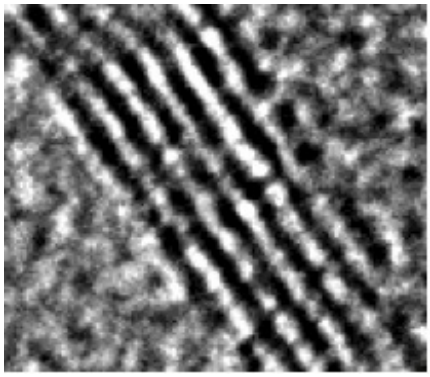




1. 384 pools of oligo probes
2. Detect signal (ds DNA)
3. Infer map & sequence

ZS Genetics

Visualization of Labeled DNA by Transmission Electronic Microscopy



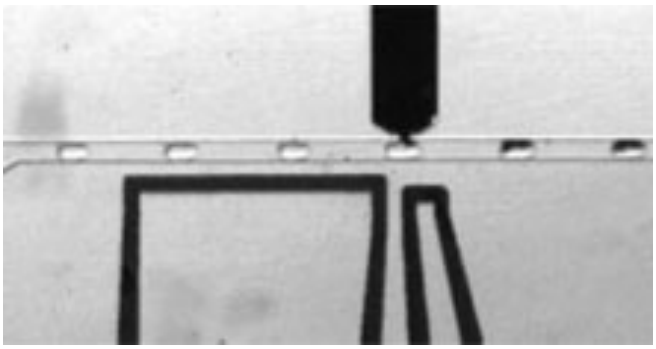
http://www.zsgenetics.com/application/GenSeq/index.html

DNA fragments labeled (iodine, bromine) during PCR amplification (labeled nucleotides)

Size and intensity differences between each labeled bases during detection in the TEM image

GnuBio

Microfluidic Manipulation of Sequencing Reactions in Picoliter Droplets



http://www.technologyreview.com/biomedicine/25481/

Optical barcodes change of color after hybridization

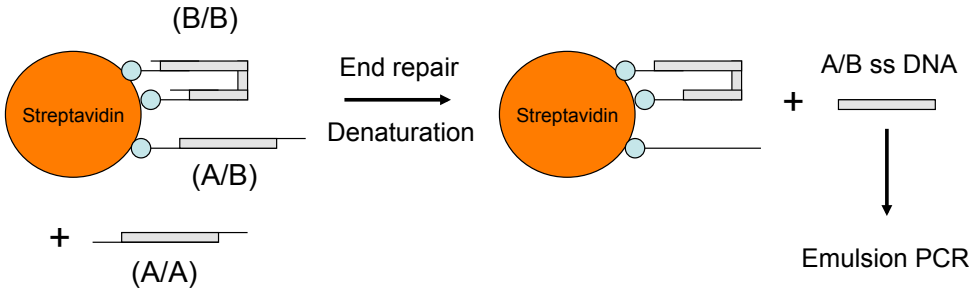
“30X Human Genome Sequencing for $30 on a 50K machine”

Questions?

Appendix

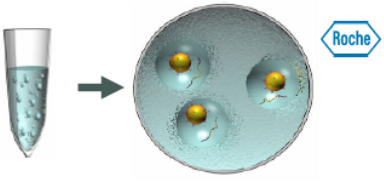
DNA Library Construction

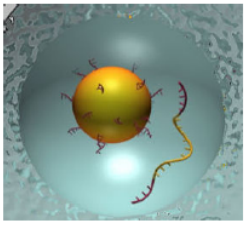
- DNA fragmentation via nebulization
- Size-selection
- Ligation of adapters A & B
- Selection of A/B fragments via biotin selection
- Denaturation to select single-stranded A/B fragments
- No cloning!



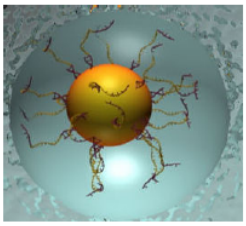
Emulsion PCR

- Add DNA to capture beads (needs titration)
- Add PCR reagents to DNA and capture beads
- Transfer sample to oil tube or cup
- Emulsify DNA capture beads in PCR reagents to form water-in-oil “microreactors”
  - Emulsion with Qiagen TissueLyser (high-speed shaker)
- Clonal amplification in microreactors
  - Careful not to break the emulsion!
  - ~10MM copies per capture bead
- Break emulsion and enrich for DNA positive beads
  - Use biotinylated oligo to capture enriched beads then denature





Before PCR



After PCR

454 SEQUENCING

www.roche-applied-science.com

Algorithms in Bioinformatics

Jim Tisdall

Programming for Biology

Lecture Notes

1. The Problem
2. Time and Space and Algorithms
3. Using Less Time
4. Using Less Space
5. Profiling
6. Parallel Processing

Suggested Reading

Mastering Algorithms with Perl
by Orwant, Hietaniemi, and Macdonald
(An excellent algorithms text with implementations in Perl)

Introduction to Algorithms
by Cormen et al.
(This is the standard modern text)

Writing Efficient Code
by Jon Bentley
(Hard to find. Great book.)

Introduction to Automata Theory, Languages, and Computation
by Hopcroft and Ullman
(The standard, mathematical textbook for theoretical computer science.)

Computers and Intractability: A Guide to the Theory of NP-Completeness
by Gary and Johnson
(Very well written.)

Network Programming with Perl
by Lincoln Stein
(Client-server network programming.)

An Introduction to Parallel Algorithms
by Joseph Jaja
(For the next generation of computers.)

Programming for Biology

Jim Tisdall, James.Tisdall -- at -- DuPont.com
Last modified: Wed Oct 14 16:14:01 EDT 2009

Time and Space and Algorithms

A program's use of time and space depends on the **algorithms** and associated **data structures** used to solve a problem.

Typically there are many *algorithms* (ways to solve a problem in a computer.) Some ways use less time and/or less space than other ways. Finding the good ways is the study of the design and analysis of algorithms.

An **algorithm** is the design or idea of a computation. It can be expressed in terms of a specific computer program, or more informally as in *pseudocode*.

A **data structure** is the form of the computation as it proceeds. A great deal of biological data is organized into **two-dimensional tables in relational databases**. Relational database tables are the standard workhorse for storing data in biology, and are useful in a surprising number of situations.

It's important to know, however, that **often the best algorithm will use some other data structure** such as a doubly-linked list or a tree, for example. Such data structures might better represent graph structures, gene networks, evolutionary relationships, and so on. And, such data structures may be used in sometimes surprising ways to speed up a computation.

The **space** of an algorithm is just the amount of computer memory it uses.

The **time** of an algorithm is usually given as a function on the size of the input. So if the input is of size n, the algorithm might take time n^2. So, for instance, if you gave such an algorithm a hundred genes, it would take about 10000 units of time to run; if you gave it ten thousand genes, it would take 100000000 units of time to run.

Time is roughly estimated according to the number of basic operations performed by your program as it runs. Basic operations are adding, concatenating two strings, printing, etc. The overall structure of the program is what is important, not an actual prediction of exactly how many seconds the program will take.

System building and knowing what can be computed

We are primarily interested in building software to achieve easily computed, but useful, results. We will not delve into the study of algorithms in any depth in this course. But it can easily happen that you may want to compute something that is hard to compute in a week, or a year, or even at all. This is a practical problem, and it's important to know what you can do about it.

The idea is that **there are limits to what can be computed.** These limits take two main forms: **intractability** and **undecidability**.

The main point:
MANY PROBLEMS CANNOT BE COMPUTED
but it's possible to get "pretty good" answers for many of them

How algorithms are measured

Algorithms are typically classified by how fast they perform on inputs of varying sizes, by giving their speed as a function of the size of the input. The size of the input is usually called **n**.

Say for example that an algorithm gets an input of size **n**, and then just to write the answer it must write an output in space of size $\mathbf{2^n}$. (The amount of space that an algorithm uses is one way to establish a lower bound for how much time the algorithm takes to complete.) Then we say the algorithm's time complexity is *"order of 2 to the n"*, written in a shorthand called **big Oh** notation as

O($\mathbf{2^n}$).

This way of measuring an algorithm is called *time complexity*.

Examples:
O($\mathbf{2^n}$) computations: *intractable* (e.g. *exponential*) is *bad*
O($\mathbf{n^2}$) computations: *tractable* (e.g. *polynomial*) is *good*
O(**5n**) computations: *tractable* (e.g. *linear*) is *great*
O(**log(n)**) computations: *tractable* (e.g. *logarithmic*) is *amazing*

If the size of the input **n** is 3, then all methods take a short amount of time -- 8 and 9 and 15 and about 1, respectively.

But if the size of the input **n = 100** , then **log(n)** is about 6, **5n** is 500, and $\mathbf{n^2}$ is 10,000 which is still not bad. However, $\mathbf{2^n}$ is bigger than the number of atoms in the universe. (And is the universe really finite? Oh well ... who's counting?)

Intractability

Intractability means that a problem cannot be computed in a reasonable amount of time. Many biological problems are intractable.

Example: in phylogeny we learn that there are many possible trees that can be built, and that the number of possible trees grows exponentially as you increase the number of taxa and as you increase the evolutionary time under discussion.

To find the best solution in an exponentially-growing space, such as the space of all possible evolutionary trees, often requires examining each possibility, and so may take an exponentially-growing time. Problems that have this property (very loosely defined here) are called
NP
(for non-deterministic polynomial time), and certain canonical such problems are called **NP-complete**.

NP-complete problems are all essentially interchangeable; that is, they all come down to essentially the same problem. The prototypical NP-complete problem is the

TRAVELING SALESMAN PROBLEM:
given a set of cities and the distances between them, what is the shortest route a traveling salesman can take to visit each one?

By the time you get to about 30 cities, the number of possible routes cannot be computed in your lifetime; by the time you reach about 60 cities, there are more possible routes than there are atoms in the universe. And we don't know a better way to find the best route than to look at each one.

An aside: no one has proved that NP-complete problems *must* require looking at each individual possibility. If you could find a *polynomial-time* algorithm for any NP-complete problem, you would be the most famous computer scientist/mathematician around, and would surely win a Nobel prize. Few people believe it will be done, but it's been an open problem for many years, and no one yet can prove that it can't be done. This is called the **P =? NP** problem.

The practical implications:

If you have a lot of data for your problem, and the problem is in NP, then you have **no practical solution to find the best, optimal answer** except on very small data sets.

But the good news is: there are **approximation algorithms** that will give you a **very good answer** in a reasonable amount of time, even if it's not the optimal answer. Such approximation algorithms underlie many of the practical approaches to such problems as phylogeny, sequence assembly, and many other problems in bioinformatics.

Undecidable problems

Less likely to be a problem for the practical bioinformatics programmer, but something to be aware of, is that there are **problems for which no solution is possible.**

These problems are called *undecidable*, and they were first demonstrated by Alan Turing and others in the 1930s.

Here's the most famous undecidable problem: the

HALTING PROBLEM
Write a program that can scan any other program and decide if it will eventually halt, or if it will go on forever without coming to a stop.

In other words, write a virus checker for nonhalting programs.

As an example of such a nonhalting "virus", here's a perl program that goes on forever (until you stop it):

```
while(1) {}
```

That looks easy to recognize. But we can *prove* that no program can be written that would catch *all* such non-halting programs.

The fact that such an easily-described problem as the HALTING PROBLEM has no solution is, when you think about it, a very deep and profound statement about the limits of human knowledge. But, nevertheless, and of a certainty, we all play on.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

Using Less Time

The Art and Science of Algorithm Design

You can divide knowledge into two types: *procedural knowledge* and *declarative knowledge*.

Declarative knowledge is a collection of facts. (E.g., Watson's great textbook "The Molecular Biology of the Gene")

Procedural knowledge is knowledge of *how to do things*, and is the kind of knowledge captured by computer algorithms. Procedural knowledge has been growing immensely since (programmable digital) computers brought the ability to specify how to do something -- that is, to formulate an *algorithm* -- to the very center of our economic, scientific, and cultural lives.

Algorithms are discovered by a combination of mathematics and art and science and luck and training and talent. Much of what we do on computers relies on the accumulated procedural knowledge -- algorithms -- of our culture.

A good algorithm is more important than a good computer

Finding a better algorithm can be much more important than getting a better, faster computer.

For the following examples I created a set of random DNA that I'll use as my "promoters". I include the code here. (We'll return to this code later in the lecture).

```
#
# Main program -- make promoters from random DNA
#

srand();

$dna = make_random_DNA(1000000);
open(DNA, ">genomic_data") or die;
print DNA $dna;

@promoters = make_random_DNA_set(10, 5000);
open(PROMOTERS, ">promoters") or die;
print PROMOTERS join("\n",@promoters),"\n";

exit 0;

#
# Subroutines
#

# Make a string of random DNA of specified length.
sub make_random_DNA {

    my($length) = @_;
    my $dna;

    for (my $i=0 ; $i < $length ; ++$i) {
        $dna .= randomnucleotide(  );
    }

    return $dna;
```

```
}

# Make a set of random DNA
sub make_random_DNA_set {

    my($length, $size_of_set) = @_;
    my $dna;
    my @set;

    # Create set of random DNA
    for (my $i = 0; $i < $size_of_set ; ++$i) {

        $dna = make_random_DNA ( $length );
        push( @set, $dna );
    }

    return @set;
}

# Select at random one of the four nucleotides
sub randomnucleotide {

    my(@nucleotides) = ('A', 'C', 'G', 'T');

    return randomelement(@nucleotides);
}

sub randomelement {

    my(@array) = @_;

    return $array[rand @array];
}
```

Consider this fragment of perl code, written to find a set of short sequences in a genome ("findpromoters0"):

```
# Read the promoter data from a file
open(PROMOTERS, "promoters") or die "a horrible death: $!";
my @promoters = <PROMOTERS>;

# Look for each occurence of each promoter in the genome
foreach my $promoter (@promoters) {
        chomp $promoter;

        # Read the genome data from a file
        open(GENOME, "genome_data") or die "a horrible death: $!";
        my $genome = <GENOME>;

        while($genome =~ /$promoter/g) {
                # $-[0] prints the location of the find
                #print "$promoter $-[0]\n"; exit;
                $db{$promoter} = $-[0];
        }
}
```

Now this code is good perl. It is syntactically correct, and it will produce the correct output. It will run, and in the end you will print out all the locations of the sequence.

Let's see how long it takes to run:

```
-bash-3.00$ date; perl findpromoters0; date
Thu Oct 20 14:28:06 EDT 2005
Thu Oct 20 14:28:48 EDT 2005
-bash-3.00$
```

Okay, so 42 seconds isn't bad! But wait ... what if we had the entire human genome, and a million tags? I'll let you do the math, or the experiment, but it takes too long.

So we try to make it faster. How? Well, we notice that for each tag, we're reading in the entire genome from the disk. Let's rewrite the code so that it only reads the genome in once (findpromoters1):

```
# Read the genome data from a file
open(GENOME, "genome_data") or die "a horrible death: $!";
my $genome = <GENOME>;

# Read the promoter data from a file
open(PROMOTERS, "promoters") or die "a horrible death: $!";
my @promoters = <PROMOTERS>;

# Look for each occurence of each promoter in the genome
foreach my $promoter (@promoters) {
        chomp $promoter;
        while($genome =~ /$promoter/g) {
                # $-[0] prints the location of the find
                #print "$promoter $-[0]\n"; exit;
                $db{$promoter} = $-[0];
        }
}
```

And the time for that is:

```
-bash-3.00$ date; perl findpromoters1; date
Thu Oct 20 14:30:46 EDT 2005
Thu Oct 20 14:31:05 EDT 2005
-bash-3.00$
```

>From 42 seconds to 19 seconds -- sweet!

But can we do better? Notice that for each promoter, we're scanning through the entire genome. So we're scanning through the entire genome 5000 times.

Is there a way we can scan through the entire genome just once? Yes, and here is one solution:

```
# Read the genome data from a file
open(GENOME, "genome_data") or die "a horrible death: $!";
my $genome = <GENOME>;

# Read the promoter data from a file
open(PROMOTERS, "promoters") or die "a horrible death: $!";
foreach $promoter (<PROMOTERS>) {
```

```
        chomp $promoter;
        $promoters{$promoter} = 1;
}

# Look for each occurence of each promoter in the genome
my $genomelength = length($genome);
for($i = 0; $i < $genomelength - 10 + 1; ++$i) {
        my $subsequence = substr($genome, $i, 10);

        # Now we just look in the hash to see if this subsequence is a promoter
        if($promoters{$subsequence}) {
                $db{$promoter} = $i;
        }
}
```

and we run a timing on it to get ("findpromoters2"):

```
-bash-3.00$ date ; perl findpromoters2 ; date
Thu Oct 20 15:42:15 EDT 2005
Thu Oct 20 15:42:16 EDT 2005
-bash-3.00$
```

That's one second, maybe less.

And so we've achieved a 43-fold speedup in our program. What was taking, say, two days to compute, now takes an hour. We couldn't have achieved that speedup going to a super expensive computer (well, maybe a cluster, which we'll discuss later.)

And so we see that finding a better algorithm is the best way to get good performance.

What, exactly, did we do? We eliminated unnecessary work. We eliminated the repetitive reading in of the genome data from the disk; and we eliminated multiple scanning through the genome data.

These are the kinds of things that you can often find in the first version of a working program. So don't neglect the important step of editing your code after you get a working draft.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

Using Less Space

Here is the main problem of space in bioinformatics:
Very large strings will swamp the main memory on your computer.

(Main memory, or RAM, is where your computer holds a running program; it is much smaller than the memory on your disks.)

When a program on your computer starts to use up too much main memory, its performance starts to degrade. The program will first enlist a portion of disk space to hold the part of the running program that it can no longer fit. This is called **swapping**.

But when a program starts swapping, which involves a lot of writing and reading to and from hard disk, it can get increasingly slow. The program may even start **thrashing**, that is, repeatedly writing and reading large amounts of data between main memory and hard disk. A program that is thrashing is going really slow, and it's slowing down the whole computer and other programs, too.

Take this snippet of code that calls `get_chromosome`:

```
my $chromosome1 = getchromosome(1);
```

When `getchromosome(1)` returns the data from human chromosome 1 to be stored in `$chromosome1`, the program uses 100Mb of memory.

Operating on the chromosome may use additional memory. For instance, in perl, when you do a regular expression search, you often want to save the successful match by using parentheses that set the special variables `$1`, `$&`, and so on.

```
$chromosome =~ /AA(GAGTC*T)/;
my $pattern = $1;
```

But once you use these special variables, the inner workings of perl require the use of considerable additional memory by your program. And you may make copies of all or part of the chromosome.

Your resulting code may be clear, straightforward to understand, and correct -- all good and proper things for code to be -- but the amount of memory usage may still seriously slow down your program.

Motto: **copying large strings is slow and takes up large amounts of memory**

Editing for Space

Often, a program that barely runs at all and takes many hours of clogging up the computer, can be rewritten to run more quickly by rewriting the algorithm so that it uses only a small fraction of the memory. It will fit into less memory, and also run a lot faster.

Use references to save space

There's one easy way to cut down on the number of big strings in a program.

Normally (without using references) a subroutine makes copies of the values passed into it, and it makes copies of the values returned from it.

References allow subroutines to avoid the string copying.

When we pass a reference to a string as an argument to a subroutine, we don't pass a copy of the string -- we

pass a reference to the string, which takes almost no additional space.

And when the subroutine ends, whatever we've done with the string is immediately available to the calling program, without having to use the `return` function, which would also copy the string.

In our example:

```
load_chromosome( 1, \$chromosome1 );
```

This new subroutine has two arguments. The `1` indicates that we want the biggest human chromosome, chromosome 1.

The second argument is a reference to a scalar variable. Inside the subroutine, the reference is most likely used to initialize an argument `$chromref`, which is a reference to the genomic data. And then, in the subroutine, the DNA data is put into the dereferenced string:

```
sub load_chromosome {
        my($chromnumber, $chromref) = @_;

        ...(omitted)...

        $$chromref = <CHROMOSOME1>
}
```

It is not necessary to return the whole chromosome from the subroutine, which would make a copy of it. The value is passed by the reference *out* of the subroutine.

Using references is also a great way to pass a large amount of data *into* a subroutine without making copies of it. In this case, however, the fact that the subroutine can change the contents of the referenced data is something to watch out for.

The rule of thumb is: **if you don't need two copies of the data, you can use references**.

Managing Memory with Buffers

One of the most efficient ways to deal with very large strings is to deal with them a little at a time.

Here is an example of a program that searches an entire chromosome for a particular 12-base pattern, using very little memory.

When searching for any regular expression in a chromosome, it's hard to see how you could avoid putting the whole chromosome in a string. But very often there's a limit to the size of what you're searching for. In this program, I'm looking for the 12-base pattern "ACGTACGTACGT."

I'm going to read the chromosome data into memory just a line or two at a time, search for the pattern, and then *reuse* the memory to read in the next line or two of data.

The extra programming work involves:

First, keeping track of how much of the data has been read in, so I can report the locations in the chromosome of successful searches.

Second, making sure I search across line breaks as well as within lines of data from the input file.

The following program reads in a FASTA file searches for my pattern in any amount of DNA--a whole chromosome, a whole genome, a year's worth of Solexa data, even all known genetic data, while using only a small amount of main memory.

```
$ perl  find_fragment  human.dna
```

For testing purposes I made a very short FASTA DNA file, `human.dna`, which contains:

```
>human dna: ACGTACGTACGT appears at positions 10, 40, and 98
AAAAAAAAAACGTACGTACGTCCGCGCGCGCGCGCGCACGTACGTACG
TGGGGGGGGGGGGGGGCCCCCCCCCCCGGGGGGGGGGGGAAAAAAAAAACG
TACGTACGTTTTTTTTTTTTTTTTTTTTTTTT
```

Here's the code for the program `find_fragment`:

```
#!/usr/bin/perl

use warnings;
use strict;

# $fragment: the pattern to search for
# $fraglen:  the length of $fragment
# $buffer:   a buffer to hold the DNA from the input file
# $position: the position of the buffer in the total DNA

my($fragment, $fraglen, $buffer, $position) = ('ACGTACGTACGT', 12, '', 0);

# The first line of a FASTA file is a header and begins with '>'
my $header = <>;

# Get the first line of DNA data, to start the ball rolling
$buffer = <>;
chomp $buffer;

# The remaining lines are DNA data ending with newlines
while(my $newline = <>) {

    # Add the new line to the buffer
    chomp $newline;
    $buffer .= $newline;

    # Search for the DNA fragment, which has a length of 12
    # (Report the character at string position 0 as being at position 1,
    # as usual in biology)
    while($buffer =~ /$fragment/gi) {
        print "Found $fragment at position ", $position + $-[0] + 1, "\n";
    }

    # Reset the position counter (will be true after you reset the buffer, next)
    $position = $position + length($buffer) - $fraglen + 1;

    # Discard the data in the buffer, except for a portion at the end
    # so patterns that appear across line breaks are not missed
    $buffer = substr($buffer, length($buffer) - $fraglen + 1, $fraglen - 1);
}
```

Here's the output of running the command
`perl find_fragment human.dna`:

```
Found ACGTACGTACGT at position 10
Found ACGTACGTACGT at position 40
Found ACGTACGTACGT at position 98
```

How the Code Works

I want to search for the fragment even if it is broken by new lines, so I'll have to look at least at two lines at a time. I get the first line, and in the while loop that follows I'll start by adding more lines to the buffer.

Then the while loop starts reading in the next lines of the FASTA file. The newline character is removed with `chomp` and the new line is added to the `$buffer`.

Then comes the short while loop that does the regular expression pattern match of the `$fragment` in the `$buffer`.

When the fragment is found the program simply prints out the fragment's position. The variable `$position` holds the position of the beginning of the buffer in the total DNA.

I also add 1, because biologists always say that the first base in a sequence of DNA is at position 1, whereas Perl says that the first character in a string is at position 0. So I add 1 to the Perl position to get the biologist's position.

The last two lines of code reset the buffer. First we eliminate the beginning (already searched) of the buffer, and then we adjust the `$position` counter accordingly. The buffer is shortened so that it just keeps the part at the very end that might be part of a pattern match that spans the newlines.

The program manages to search the entire genome for the fragment, while keeping at most two lines' worth of DNA in `$buffer`, It performs very quickly, compared to a program that reads in a whole genome and blows out the memory in the process.

When You Should Bother

Programs may be developed on one computer, but run on very different computers.

A space-inefficient program might well work fine on your computer, but not work well at all when you run it on another computer with less main memory installed. Or, it might work fine on the fly genome, but start thrashing when you try it on the human genome.

If you know you'll be dealing with large data sets, like genomes, take the amount of space your program uses as an important constraint when designing and coding. Then you won't have to go back and redo the entire program when a large amount of DNA gets thrown at the program.

Data Compression

In Perl, as in any programming system, the size of the data that the program uses is an absolute lower bound on how fast the program can perform.

Each base is typically represented in a computer language as one ASCII character taking one 8-bit byte, so 3 gigabases equals 3 gigabytes. Of course, you could represent each of the four bases using only 2 bits, so considerable compression is possible; but such space efficiency is not commonly employed. When it is, you can pack 4 times as much data into a given space (for nucleotides, that is.)

```
A 00
C 01
G 10
T 11
```

If you want an exercise, try using perl functions `pack` and `vec` to compress DNA sequence data to 4 bases per byte.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008

Profiling

You saw earlier an easy way on Unix to see how long a program takes:

```
date; perl findpromoters1; date
```

This prints the time, then immediately runs the program, and then immediately prints the time again.

Perl has several much more detailed ways to examine the performance of a program.

I'll just show you one of them, called **`DProf`**. `DProf` reports on various aspects of your program's performance.

The most valuable report is probably the summary by subroutine.

By seeing which subroutines are taking the most time, you can narrow your re-editing of the program to just those subroutines, and quickly make the improvements where they count the most.

For demonstration, I'm going to use a program with a few subroutines; namely, the `makerandom` program we used earlier to make random DNA genomic sequence and putative DNA binding sites.

First you have to load the `Devel::Prof` module in your program. You do this by adding the `-d:DProf` command-line argument. Then when your program runs, the module makes counts of many things in the program. Your program will take a bit longer to run, but you'll collect valuable statistics on its performance.

So one can simply run the program as usual, adding the command-line argument. When it's done, it will have created a file called `tmon.out` in my directory. I then run the `dprofpp tmon.out` program to see the results of the profile of my program:

```
$ perl -d:DProf makerandom
$ dprofpp tmon.out
Total Elapsed Time = 5.464274 Seconds
  User+System Time = 5.354274 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 72.2   3.870  7.594 105000   0.0000 0.0000  main::randomnucleotide
 69.5   3.725  3.725 105000   0.0000 0.0000  main::randomelement
 33.7   1.807  9.402   5001   0.0004 0.0019  main::make_random_DNA
 0.22   0.012  0.525      1   0.0125 0.5250  main::make_random_DNA_set
$
```

If I wanted to speed this program up, I'd head straight for the `randomelement` and `randomnucleotide` subroutines to see what I might be able to tweak in them, since my analysis shows that they take almost all the time in the program.

`DProf` has many options, but this is how I almost always use it, as it's simple and tells me what I need to know.

Some older perls might not have `DProf` installed, in which case you have to do something like this: (you may need root permission):

```
$ perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support enabled

cpan> install Devel::DProf
CPAN: Storable loaded ok
Going to read /root/.cpan/Metadata
  Database was generated on Wed, 19 Oct 2005 22:01:03 GMT
Devel::DProf is up to date.

cpan> quit
Lockfile removed.
$
```

In this case perl reported that the `Devel::DProf` module was already installed with the latest version; if not, it would have installed it.

You know, I wonder if I can speed up my `makerandom` program. Let's look at it. Hmmm. I did try a few things out: let's see how the new program `makerandom2` behaves:

```
$ perl -d:DProf makerandom2
$ dprofpp tmon.out
Total Elapsed Time =  1.27999 Seconds
  User+System Time =  1.27999 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 96.8   1.240  1.240   5001   0.0002 0.0002  main::make_random_DNA
 0.78   0.010  0.050      1   0.0100 0.0500  main::make_random_DNA_set
$
```

Cool! From over 5 seconds to a little over 1 second. A five-fold speedup!

How did I do it? Here's the new version:

```
srand();

my(@nucleotides) = ('A', 'C', 'G', 'T');

$dna = make_random_DNA(1000000);
open(DNA, ">genomic_data") or die;
print DNA $dna;
```

```
@promoters = make_random_DNA_set(10, 5000);
open(PROMOTERS, ">promoters") or die;
print PROMOTERS join("\n",@promoters),"\n";

# Make a string of random DNA of specified length.
sub make_random_DNA {

    my($length) = @_;
    my $dna;

    for (my $i=0 ; $i < $length ; ++$i) {
        $dna .= $nucleotides[rand @nucleotides];
    }

    return $dna;
}

# make_random_DNA_set
sub make_random_DNA_set {

    my($length, $size_of_set) = @_;
    my $dna;
    my @set;

    # Create set of random DNA
    for (my $i = 0; $i < $size_of_set ; ++$i) {

        # make a random DNA fragment
        $dna = make_random_DNA ( $length );

        # add $dna fragment to @set
        push( @set, $dna );
    }

    return @set;
}
```

First, I moved the line

```
my(@nucleotides) = ('A', 'C', 'G', 'T');
```

out of a subroutine and up to the top of the program. This way the array doesn't have to get reinitialized each time the program is called.

But much more importantly, I eliminated two subroutine calls entirely, and put their functionality directly into the lines of code that were calling them. First I axed `randomelement` by putting its functionality directly into the calling subroutine `randomnucleotide`: from

```
sub randomnucleotide {

    my(@nucleotides) = ('A', 'C', 'G', 'T');

    return randomelement(@nucleotides);
}

sub randomelement {

    my(@array) = @_;

    return $array[rand @array];
}
```

to

```
my(@nucleotides) = ('A', 'C', 'G', 'T');

sub randomnucleotide {

    return $nucleotides[rand @nucleotides];
}
```

and finally I eliminated `randomnucleotide` by putting its code directly into the calling program: from

```
        $dna .= randomnucleotide(  );
```

to

```
        $dna .= $nucleotides[rand @nucleotides];
```

In short, I eliminated two subroutine calls that were each being called 105000 times, and that made a significant speedup. Usually, you're more likely to try to improve a subroutine than to eliminate it, but as you see eliminating a subroutine can on occasion have big payoffs.

The book by Bentley "Writing Efficient Code" discusses such "tricks" in entertaining and useful detail.

So I hope you're convinced that `DProf` is worthwhile. There are other profiling methods available in Perl too, and you might want to explore them.

There are different ways to think of parallel processing.

Parallel Algorithms

One kind of parallel processing actually uses the specific topology of the interconnections between the CPUs to implement new kinds of algorithms. This kind of parallel processing is fascinating and gives you very fast programs, but is *way* beyond the scope of this lecture or this course. But I thought you'd like to know that it exists.

In this hard-core parallel algorithms work, you might work on special computers (e.g. "grids", "butterfly networks") or even on purely theoretical models of parallel computation, and you design algorithms to run on those types of parallel computers.

Parallel Processing on Networks and Clusters

More common is this scenario: say you are doing 40 tasks, one after the other, and each one takes an hour. It will take your working week to finish the tasks.

Now let's say you figure out a way to do all the tasks simultaneously, and each one still takes an hour. You'll now finish the tasks, all of them, in one hour instead of one week.

One kind of parallel processing is just like this example. That's the kind of parallelism I'll talk about here, in terms of networks and clusters and threads. You simply divide your program up into parts that can be performed simultaneously, and then you run each part on its own CPU. Not all problems can be divided up like this, but those that can (say running a million blast searches) can get big speedups fairly easily.

Network Programming

One of the most successful forms of multi-processor computing has been *network programming*.

Network programming involves connecting two or more computers by a communications line and implementing a protocol that enables them to exchange information.

The development of computer networks began in earnest in the 1950s, and the various networks were interconnected by the *internet* (from *inter*connected *net*works) beginning in the late 1970s.

The protocols supported by the internet gradually expanded, until the protocols known as the *web* (or "world wide web") became widely popular beginning around 1990.

It is quite possible to program several computers to interact, using the several programming interfaces to the protocols that are available from such languages as perl.

Perl has supported these protocol interfaces since the beginning. I can speak from personal experience that it's a lot of fun to build a useful network service in this way. (In 1992 I was searching all of Genbank with regular expressions in about 35 seconds, by distributing the job with a network service written entirely in perl.)

I recommend the book "Network Programming with Perl" by Lincoln Stein if you're interested in these techniques.

Threads

Threads are different from, but related to, multiprocessing. Threads are multiple execution paths built into one process, that share resources like global variables, signals, and such. You can have a multithreading program that runs on a single processor; or, if you're running on a multiprocessor (it's common to have from 2 to around 24 processors on a given machine) the threads may be executed on different processors, giving you the advantage of parallelism.

Threads are a capability that is built into an operating system (or not, as the case may be.) If your operating system supports threads, and your programming language gives you access to them, then you can use them in your program.

If you're interested in threads, you want to use the "threads" (not "Threads") module:

```
use threads;
```

I'm going to skip the examples of threads programs: see me if you're interested.

Clusters

Clusters are multiple CPUs joined in a simple network. They are typically used to take a program that must compute the same way over many inputs, and run the program on all the CPUs, dividing the input up between them.

If you have access to a (usually) Linux cluster where you work, take the time to find out how to submit programs to it.

In a recent job I had, I had to do three computation-intensive calculations over several genomes. Each one took a week or two to finish when running on a single computer. On the Linux cluster, they all finished within a small number of hours, and using that precomputation I was able to carry my search for novel genes to a successful conclusion.

This Linux cluster has about 450 CPUs, and is a fairly big one. But it's quite straightforward -- you could do it yourself -- to buy 10 or 20 inexpensive Linux boxes and construct a Linux cluster that can speed up your large-scale, repetitive computations by 10 or 20 times.

Using Less Space

Here is the main problem of space in bioinformatics:
Very large strings will swamp the main memory on your computer.

(Main memory, or RAM, is where your computer holds a running program; it is much smaller than the memory on your disks.)

When a program on your computer starts to use up too much main memory, its performance starts to degrade. The program will first enlist a portion of disk space to hold the part of the running program that it can no longer fit. This is called **swapping**.

But when a program starts swapping, which involves a lot of writing and reading to and from hard disk, it can get increasingly slow. The program may even start **thrashing**, that is, repeatedly writing and reading large amounts of data between main memory and hard disk. A program that is thrashing is going really slow, and it's slowing down the whole computer and other programs, too.

Take this snippet of code that calls `get_chromosome`:

```
my $chromosome1 = getchromosome(1);
```

When `getchromosome(1)` returns the data from human chromosome 1 to be stored in `$chromosome1`, the program uses 100Mb of memory.

Operating on the chromosome may use additional memory. For instance, in perl, when you do a regular expression search, you often want to save the successful match by using parentheses that set the special variables `$1`, `$&`, and so on.

```
$chromosome =~ /AA(GAGTC*T)/;
my $pattern = $1;
```

But once you use these special variables, the inner workings of perl require the use of considerable additional memory by your program. And you may make copies of all or part of the chromosome.

Your resulting code may be clear, straightforward to understand, and correct -- all good and proper things for code to be -- but the amount of memory usage may still seriously slow down your program.

Motto: **copying large strings is slow and takes up large amounts of memory**

Editing for Space

Often, a program that barely runs at all and takes many hours of clogging up the computer, can be rewritten to run more quickly by rewriting the algorithm so that it uses only a small fraction of the memory. It will fit into less memory, and also run a lot faster.

Use references to save space

There's one easy way to cut down on the number of big strings in a program.

Normally (without using references) a subroutine makes copies of the values passed into it, and it makes copies of the values returned from it.

References allow subroutines to avoid the string copying.

When we pass a reference to a string as an argument to a subroutine, we don't pass a copy of the string -- we

pass a reference to the string, which takes almost no additional space.

And when the subroutine ends, whatever we've done with the string is immediately available to the calling program, without having to use the `return` function, which would also copy the string.

In our example:

```
load_chromosome( 1, \$chromosome1 );
```

This new subroutine has two arguments. The `1` indicates that we want the biggest human chromosome, chromosome 1.

The second argument is a reference to a scalar variable. Inside the subroutine, the reference is most likely used to initialize an argument `$chromref`, which is a reference to the genomic data. And then, in the subroutine, the DNA data is put into the dereferenced string:

```
sub load_chromosome {
        my($chromnumber, $chromref) = @_;

        ...(omitted)...

        $$chromref = <CHROMOSOME1>
}
```

It is not necessary to return the whole chromosome from the subroutine, which would make a copy of it. The value is passed by the reference *out* of the subroutine.

Using references is also a great way to pass a large amount of data *into* a subroutine without making copies of it. In this case, however, the fact that the subroutine can change the contents of the referenced data is something to watch out for.

The rule of thumb is: **if you don't need two copies of the data, you can use references**.

Managing Memory with Buffers

One of the most efficient ways to deal with very large strings is to deal with them a little at a time.

Here is an example of a program that searches an entire chromosome for a particular 12-base pattern, using very little memory.

When searching for any regular expression in a chromosome, it's hard to see how you could avoid putting the whole chromosome in a string. But very often there's a limit to the size of what you're searching for. In this program, I'm looking for the 12-base pattern "ACGTACGTACGT."

I'm going to read the chromosome data into memory just a line or two at a time, search for the pattern, and then *reuse* the memory to read in the next line or two of data.

The extra programming work involves:

First, keeping track of how much of the data has been read in, so I can report the locations in the chromosome of successful searches.

Second, making sure I search across line breaks as well as within lines of data from the input file.

The following program reads in a FASTA file searches for my pattern in any amount of DNA--a whole chromosome, a whole genome, a year's worth of Solexa data, even all known genetic data, while using only a small amount of main memory.

```
$ perl  find_fragment  human.dna
```

For testing purposes I made a very short FASTA DNA file, `human.dna`, which contains:

```
>human dna: ACGTACGTACGT appears at positions 10, 40, and 98
AAAAAAAAAACGTACGTACGTCCGCGCGCGCGCGCGCACGTACGTACG
TGGGGGGGGGGGGGGGCCCCCCCCCCCGGGGGGGGGGGGAAAAAAAAAACG
TACGTACGTTTTTTTTTTTTTTTTTTTTTTTT
```

Here's the code for the program `find_fragment`:

```
#!/usr/bin/perl

use warnings;
use strict;

# $fragment: the pattern to search for
# $fraglen:  the length of $fragment
# $buffer:   a buffer to hold the DNA from the input file
# $position: the position of the buffer in the total DNA

my($fragment, $fraglen, $buffer, $position) = ('ACGTACGTACGT', 12, '', 0);

# The first line of a FASTA file is a header and begins with '>'
my $header = <>;

# Get the first line of DNA data, to start the ball rolling
$buffer = <>;
chomp $buffer;

# The remaining lines are DNA data ending with newlines
while(my $newline = <>) {

    # Add the new line to the buffer
    chomp $newline;
    $buffer .= $newline;

    # Search for the DNA fragment, which has a length of 12
    # (Report the character at string position 0 as being at position 1,
    # as usual in biology)
    while($buffer =~ /$fragment/gi) {
        print "Found $fragment at position ", $position + $-[0] + 1, "\n";
    }

    # Reset the position counter (will be true after you reset the buffer, next)
    $position = $position + length($buffer) - $fraglen + 1;

    # Discard the data in the buffer, except for a portion at the end
    # so patterns that appear across line breaks are not missed
    $buffer = substr($buffer, length($buffer) - $fraglen + 1, $fraglen - 1);
}
```

Here's the output of running the command
`perl find_fragment human.dna`:

```
Found ACGTACGTACGT at position 10
Found ACGTACGTACGT at position 40
Found ACGTACGTACGT at position 98
```

How the Code Works

I want to search for the fragment even if it is broken by new lines, so I'll have to look at least at two lines at a time. I get the first line, and in the while loop that follows I'll start by adding more lines to the buffer.

Then the while loop starts reading in the next lines of the FASTA file. The newline character is removed with `chomp` and the new line is added to the `$buffer`.

Then comes the short while loop that does the regular expression pattern match of the `$fragment` in the `$buffer`.

When the fragment is found the program simply prints out the fragment's position. The variable `$position` holds the position of the beginning of the buffer in the total DNA.

I also add 1, because biologists always say that the first base in a sequence of DNA is at position 1, whereas Perl says that the first character in a string is at position 0. So I add 1 to the Perl position to get the biologist's position.

The last two lines of code reset the buffer. First we eliminate the beginning (already searched) of the buffer, and then we adjust the `$position` counter accordingly. The buffer is shortened so that it just keeps the part at the very end that might be part of a pattern match that spans the newlines.

The program manages to search the entire genome for the fragment, while keeping at most two lines' worth of DNA in `$buffer`, It performs very quickly, compared to a program that reads in a whole genome and blows out the memory in the process.

When You Should Bother

Programs may be developed on one computer, but run on very different computers.

A space-inefficient program might well work fine on your computer, but not work well at all when you run it on another computer with less main memory installed. Or, it might work fine on the fly genome, but start thrashing when you try it on the human genome.

If you know you'll be dealing with large data sets, like genomes, take the amount of space your program uses as an important constraint when designing and coding. Then you won't have to go back and redo the entire program when a large amount of DNA gets thrown at the program.

Data Compression

In Perl, as in any programming system, the size of the data that the program uses is an absolute lower bound on how fast the program can perform.

Each base is typically represented in a computer language as one ASCII character taking one 8-bit byte, so 3 gigabases equals 3 gigabytes. Of course, you could represent each of the four bases using only 2 bits, so considerable compression is possible; but such space efficiency is not commonly employed. When it is, you can pack 4 times as much data into a given space (for nucleotides, that is.)

```
A 00
C 01
G 10
T 11
```

If you want an exercise, try using perl functions `pack` and `vec` to compress DNA sequence data to 4 bases per byte.

Programming for Biology Last modified: Mon Oct 20 23:14:38 EDT 2008