Perl in a Day
Peeking Inside the Oyster

Biology-Flavored Perl Overview

Amir Karger – `amir_karger@hms.harvard.edu`
Research Computing, HMS
rc.hms.harvard.edu/training

Class Overview

- Introduction – Why learn Perl?
- Scripting – Reproducible Science
- Variables – Making Programs Reusable
- Control Structures – Doing Things Lots of Times (Or Not)
- Data Munging – Perl for Bioinformatics
- Arrays and Hashes – Groups of Things
- Subroutines & Modules – Making Programs *Really* Reusable
- Objects – Complex Data, Complex Workflow
- BioPerl – Doing (More) Bio With Perl

Research Computing Commercial

- Knowledge + experience in science + computers
  - We worry about computers so you can do science
  - Backup, installation, security, scripting…
- Wiki and more: http://rc.hms.harvard.edu
- Tickets
  - Research questions: rchelp@hms.harvard.edu
  - Other questions: “Support” on http://it.med.harvard.edu
  - The more detail, the better
- Talk to us **before** you do lots of work
  - Save time
  - Do better science

While I'm Talking…

- If you're running Windows, reboot to MacOS
- Browse to http://rc.hms.harvard.edu/training
  - Click "perl"
  - Download PDF slides. Download and unzip unixcode.zip
- Mac: use TextEdit (search for it in)
  - (Just the first time)
  - Close the first editing window it opens
  - Go to TextEdit->Preferences
  - Select "Save as Plain text" (or Perl programs won't run)
- Open a Terminal (search for it in)
  - cd Downloads; cd unixcode; cd exercises_UNIX

While I'm Talking…

- Browse to http://rc.hms.harvard.edu/training
  - Click "perl"
  - Download PDF slides.
- Raise your hand if you need an Orchestra account
- Talk to a TA if you need a laptop
- Browse to the “Logging into Orchestra slide”
  - Try logging in

The Bad News

- You can't learn programming in such a short time
  - Too much syntax
  - Too many functions
  - Too many concepts
  - Too many special cases (especially in Perl)

The Good News

- You can do a lot knowing just a little Perl
- Perl is good at bioinformatics
- Perl is fun!

Objectives

- Understand the basics of Perl
  - Focus on what kinds of things Perl can do
  - Don't worry too much about syntax
- Learn to read, modify, and run Perl scripts
- Learn some mistakes to avoid
- Answer your questions (maybe after class)
- Special focus on *data munging*
  - Data what?

Data Munging

- Doing stuff with data
  - Getting data from many sources
    - Keyboard, local files, databases, ftp, web, …
  - Reading (and understanding) data
    - Binary, Text, HTML, XML, zip, Graphics, …
    - BIG files, many files
  - Combining data
  - Analyzing data (e.g., mathematically)
  - Filtering data
  - Outputting data
- Lots of bioinformatics is just data munging
- Perl is very (very) good at data munging

Why Perl?

- Easy to learn and quick to write
  - Rapid prototyping
  - But scalable to large programs
- Kitchen sink language
  - Combines parts of many other tools (C, sed, awk, sh, …)
  - Call other programs
- Cross-Platform: Windows, Mac, UNIX
- Open Source – lots of code already available
- TMTOWTDI - There's more than one way to do it
- Very popular in Bioinformatics

What Makes Perl Different?

- More like English, less like Math
  - (Pluses or minuses…)
  - More messy (writing vs. reading)
  - Less orthogonal (TMTOWTDI vs. inelegance)
  - Huge set of libraries available (which is best?)
  - Regular expressions (power vs. complexity)
  - Interpreted, not compiled (fast writing vs. running)
  - DWIM – "Do what I mean" (convenience vs. confusion)

Why Not Perl? (A Biased View)

- Perl is not the fastest-running language
  - Not good for doing huge amounts of very complex math
  - But you often save time by developing code quickly
- Perl allows you to write messy code
  - "Write-only language"
  - But messy is fine in certain contexts
  - Perl can help you write clean code
- Not originally designed for huge programs
  - Older versions of Perl made it hard
  - But plenty of huge programs have been written in Perl
  - This class isn't for people writing huge programs

What Can Perl Do for Me?

- Automate other programs
  - Run 1,000 BLASTs
  - High-throughput downloading and analysis of biological databases
- Analyze, filter, merge, reformat data
  - Munge results of other programs
  - Write one-liners to explore your data
- Interact with SQL databases (MySQL, Oracle, etc.)
  - Store, read, change structured data
- Create interactive CGI web pages
  - UCSC, BLAST, a simple login form
- Other bioinformatics topics
  - Population Genetics, Ontologies, Alignments, Graphing, …

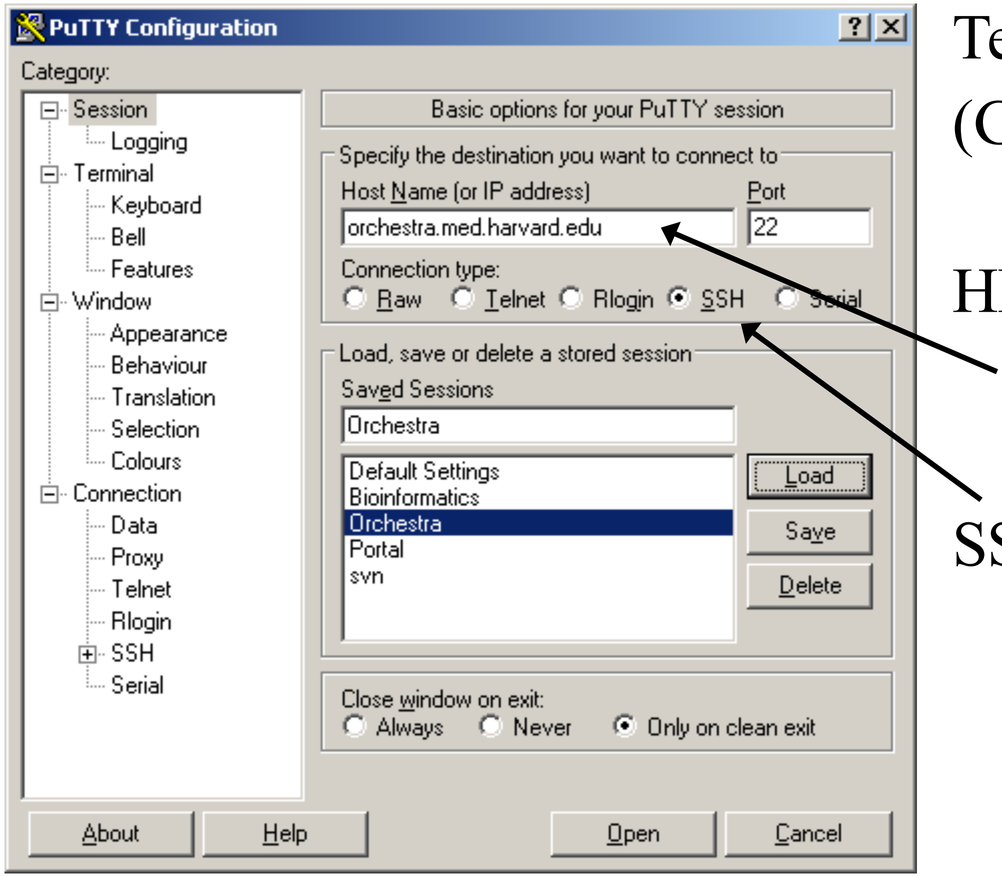
Getting Started

- Where is Perl?
  - On any UNIX (Linux, Mac) computer
  - On the HMS cluster (orchestra.med.harvard.edu)
  - On the FAS cluster (odyssey.fas.harvard.edu)
  - Windows: download from http://www.activestate.com/Products/ActivePerl
- Don't run on your own laptop!
  - Unless you have BLAST+ installed

Logging into Orchestra from Mac (or Linux)

- Open a (Mac) Terminal window
  - Search for it in if it isn't on your desktop
- **`ssh ab123@orchestra.med.harvard.edu`**
  - Replace ab123 above with your eCommons

Logging into Orchestra from Windows


PuTTY Configuration

Category:

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): orchestra.med.harvard.edu

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions: Orchestra

Default Settings
Bioinformatics
Orchestra
Portal
svn

Close window on exit: Always Never [✓ Only on clean exit]

Terminal program: putty.exe (Google Putty and SSH)

HMS cluster "head" node: orchestra.med.harvard.edu

SSH, a secure telnet. (Port will change to 22)

Getting Ready to run Perl on Orchestra

- WARNING! Cutting and pasting may break!
  - Dashes may not be dashes, quotes can break, etc.
- Get an interactive shell
  - **`bsub -q interactive -Is bash`**
- Find Perl and Bioperl on Orchestra
  - (Needed if you want Bioperl or other Perl libraries)
  - **`module load dev/perl/5.18.1`**
- Find BLAST
  - (Needed if you're using BLAST)
  - **`module load seq/blast/ncbi-blast/2.2.30`**

Getting the Sample Code (Orchestra)

- Get the zipped code
  - `cp /groups/rc-training/perl/unixcode.zip ./`
- Unzip code
  - `unzip unixcode.zip`
- Change to the sample directory and run a program
  - `cd unixcode`
  - Class demos are in `class_demos_UNIX`, etc.
  - `cd exercises_UNIX`
  - `perl EX_Scripting_1.pl`
- List of programs in class order (in demo directory)
  - `more MANIFEST`

Getting the Sample Code (UNIX/Mac)

- Get the zipped code
  `http://rc.hms.harvard.edu/training/perl/unixcode.zip`
- Open a Terminal window
- Unzip code
  - **`unzip unixcode.zip`**
- Change to the sample directory
  - **`cd Downloads`**
  - **`cd unixcode`**
  - Class demos are in `class_demos_UNIX`, etc.
  - **`cd exercises_UNIX`**
- List of programs in class order (in demo directory)
  - **`more MANIFEST`**

Before you start using Perl…

- Make sure Perl exists, find out what version it is
  - **`perl -v`**
- How do I get help?
  - **`perldoc perl (general info, TOC)`**
  - **`perldoc perlop (operators like +, *)`**
  - **`perldoc perlfunc (functions like chomp: > 200!)`**
  - **`perldoc perlretut (regular expressions: /ABC/)`**
  - **`perldoc perlreref (regular expression reference)`**
  - **`perldoc -f chomp (what does chomp function do?)`**
  - **`perldoc Getopt::Long (learn about a Perl module)`**
- Type q to quit when viewing help pages,
- Space bar for next page

Editing your files graphically

- Writing and running Perl programs
  - Use any text editor to edit a program
  - Save as whatever.pl in the correct directory
  - Run from the command line with `perl whatever.pl`
- Mac: use TextEdit (search for it in 🔍)
  - See "While I'm Talking" slide
  - If you see a ruler in your doc, Format->Make Plain Text
  - Save files in Downloads/unixcode/exercises_UNIX
- Windows: http://winscp.net edits remote files
  - Notepad or Wordpad to edit local files

Editing your files with nano

- Use an editor to write your programs
  - pico, emacs, vi (or vim) are some UNIX options
  - Type `nano blah.pl` to edit a new or existing file
- Type your program
  - "Enter" to start a new line
  - Arrow keys, not mouse, to move around
- Common commands at bottom of screen
  - Control-O   Save   (Not Control-S!)
  - Control-X   Quit

Exercise – "Hello, World!"

```
print "Hello, World!\n"; # A comment
```

- Type the above program (one line) into TextEdit
- Save as Downloads/unixcode/exercises_UNIX/hello.pl
- Save as PLAIN text (not rich text) without .txt
- Run it from the Terminal

```
% perl hello.pl
Hello, World!
```

You have written your first Perl program!

First Perl Program

Comment - any text after # sign - doesn't do anything

```
# Hack into government computers...
```

\n makes a new line (inside "double quotes")

```
print "Hello, World!\n";
```

Many Perl scripts start with a #! line

- For now, ignore this
- The -w is like typing "use warnings"

```
#!perl -w     (or maybe #!/usr/bin/perl -w)
```

First Perl Program II

```
print "Hello, World!\n"; # A comment
```

- "**;**" is used at the end of each command
  - A command is *usually* one line
  - But multi-line commands, multi-command lines OK
  - Semicolons are (sometimes) optional
- Warning: Perl is case sensitive!
  - print is not the same as Print
  - $bio is not the same as $Bio

First Perl Program III

- `print` is a *function* which prints to the screen
  - `print("Hi")` is (usually) the same as `print "Hi"`
  - Inside "double quotes", \n starts new line, \t prints tab
  - A function is *called* with zero or more *arguments*
    - Arguments are separated by commas
    - `print` takes as many arguments as you give it

```
print ""; # legal, prints nothing, not even \n
print("Hi", "There"); # prints HiThere
print(Hi); # illegal (calls the function Hi)
print(1+1, 2+2, "\n"); # prints 24 and a newline
```

Scripting

Reproducible Science
via
Scripting command-line calls

3-Minute Introduction to Biology

- BLAST program
  - Finds DNA or protein sequences similar to your **query** sequence(s)
  - Better results have lower **E-value** (e.g., 1e-5 is better than .03)
  - Our results will be in tabular format

```
Query_id123   Subject_id456   92.20   510   86   1  602   1101   29   560   1e-20  459
```

  - A **hit** means part or all of the **query** sequence is similar to a **subject** sequence in the big search database
- FASTA file format
  - Standard format for storing DNA or protein sequences
  - Identifier, (optional) description, sequence

```
>blah|12345    Cytochrome c oxidase
ACTGGTCGAAGTTGGCGA
ACGGTTGGTACGCA
```

- Examples are biology-specific, but the Perl ideas aren' t

Embedding Shell Commands

Use shell commands in Perl programs:

```
system("ls"); # list files in current directory
```

Run a BLAST with tabular output:

```
system("blastn -task blastn -db fungi -query one_seq.fasta
     -outfmt 6 -evalue 1e-4 > one_seq.blast");
```

Search for text string “Cgla” in BLAST output file:
(UNIX, Mac, Cygwin in Windows. No Cygwin? Use “find”)

```
system("grep 'Cgla’ one_seq.blast”);
```

Embedding shell commands II

Multiple commands in sequence → script

```
# Blast a yeast sequence against many fungi
system("blastn ... > one_seq.blast");

# Find Candida glabrata hits
system("grep 'Cgla' one_seq.blast");
```

Benefits over running from command line:

- Easy to repeat (reproducible science)
- Easy to rerun with slightly different parameters

Easier if parameters are at the top of the program

- … or program asked us for them

Exercise – Automate BLAST and grep

1. Run the script to BLAST and grep
   - `perl EX_Scripting_1.pl`
2. Now edit `EX_Scripting_1.pl` and change the way you're BLASTing and greping.
   a) How many Sklu hits are there?
   b) How many Kwal hits?
   c) BLAST with 1e-50 instead of 1e-4
      How many Cgla hits do you get now?

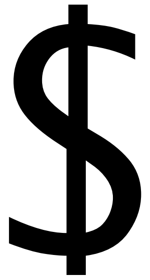
- Exercises are in exercises_UNIX/
- Solutions are in solutions_UNIX/
  Look for "# CHANGED" lines

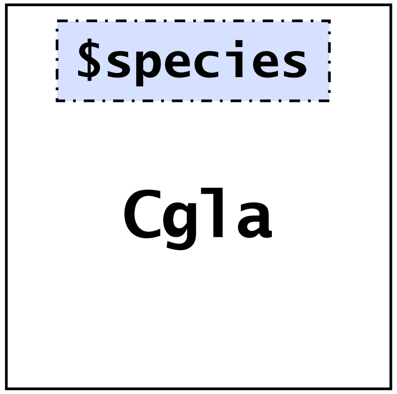
Variables

Making Programs Reusable
by
Storing and Manipulating Data

Scalar Variables

- A box containing a single “thing” (value)
  - `$e_value = 1e-4;`
  - `$string = "has spaces and $vars and \n";`
  - `$species = "";`
  - References, objects
- Has a name (label) starting with $
- Value can change during a program
  - `$species = "Cgla";`
  - `$species = "Ylip"; # later…`
- Variables encourage reusability
- See **`variables.pl`**





Scalar Variables II – Declaring Variables

- Declare variables with **`my`**
  - Tell the program there's a variable with that name
  - **`my $e_value = 1e-4;`**
  - Use **`my`** the first time you use a variable
  - Don't have to give a value (default is "", but –w may warn)
- Avoid typos
  - **`use strict;`**
  - Put this at the top of (almost) any program
  - Now Perl will complain if you use an undeclared variable
  - **`$evalue = 1e-10; # "Global symbol…"`**
- Better to get parameters from the user…

Reading Variables – From the Keyboard

- See **`variables_ask.pl`**
- Use `<>` to read in a line of input from the keyboard
  - `$species = <>;`
  - Result gets placed in variable `$species`
  - Typing Cgla and Enter yields same results as this code:

```
$species = "Cgla\n";
```

- `chomp()` removes the newline (\n) from the input
  - `$species` is now `Cgla`
  - `chomp()` only removes a newline
  - `chomp()` only removes newline at the end of a string

Reading Variables – From an Input File

- `<>` can also read from input files
  - Specify input file(s) on the command line
  - `perl variables_ask.pl variables_ask.in`
  - Use `<>` for multiple files of the same type
    - E.g., Multiple BLAST outputs, or multiple FASTA files
  - `<>` reads data from files as if you typed it on the keyboard
- Saving input files → Reproducible Science!
- But this is a lot of work, for one or two options…

Reading Variables – From the Command Line

- Getopt::Long
  - A module (library of functionality someone else wrote)
  - Allows you to input simple options on the command line
  - **`perldoc Getopt::Long`** for (much) more information
- Using Getopt::Long
  - `use Getopt::Long; # use the module`
  - `my $species = "Cgla"; # default value for variable`
  - `GetOptions("spec=s" => \$species);`
  - `spec` means you can type -spec, -sp, -s on command line
  - `=s` means text **s**tring (`=i` for **i**nteger, `=f` for "**f**loat" decimal)
  - `=>` is a fancy comma
  - `\$species` is a "reference" (pointer) to `$species` variable

Reading Variables – From the Command Line II

- See **`get_s_opt.pl`**
  - Run it like this: **`perl get_s_opt.pl -s "Klac"`**
  - **Not** like this: **`perl -s "Klac" get_s_opt.pl`**
  - If also giving files: **`perl get_s_opt.pl -s "Klac" file1`**
- You can input multiple parameters
  - Call **`GetOptions`** only *once* near beginning of program
  - Tell **`GetOptions`** about all possible options
  - **`GetOptions(`**
    - **`"spec=s" => \$species,`**
    - **`"blast=s" => \$run_blast`**
  - **`);`**
  - **`GetOptions`** will set **`$species`** and **`$run_blast`** (if user inputs -blast and -spec)

Getting output from shell commands

- Use backquotes (``) around shell command
- Runs the command (like `system()`)
- Gets the results in a variable
  - You get the standard output, i.e., what would have been printed to the screen
    - (But standard error will still print to the screen)
  - You can embed $variables in the command

```
$date = `date`; # UNIX command: guess what it does?
print "The date is $date";
# Note: returns a LONG string with \n's in it!
$blast = `blastn –task blastn –evalue $e_value …`
```

Exercise – Variables and Inputting Options

1. Input the E-value to use for BLAST from the user
   - Change `EX_Variables_1.pl`
   - Input E-value from the keyboard (*before* BLASTing!)
   - Using same program, input from a file (with two lines)
   - Input from two separate, one-line files. (Type file names in the right order!)
2. Use Getopt::Long
   - Start with `EX_Variables_2.pl`
   - Add –evalue parameter
   - E-value is a "float" (decimal number); use **`=f`**, not **`=s`**

Control Structures

Doing Things Lots of Times (Or Not)
using
Loops and Conditions

Loops and Conditions – Why?

- So far we have seen only linear programs
- Flowcharts are more interesting (and realistic)
  - Loops - do something more than once
  - Conditions - do something sometimes, but not other times

- Combining loops and conditions correctly is a major part of programming

Conditions

- Let's stop running BLAST every time
- Basic if statement:
  - `if (condition)` is true…
  - Run `{BLOCK}` of code

```
if (condition) {
    do some stuff;
    and more stuff;
}
```

```
if ($run_blast eq "y") {
    my $note = "let's rerun!";
    print "$note\n";
    system("blastn …");
}
print $note; # ERROR. Unknown var
```

- No semicolon after beginning and end braces
- Blocks are often indented for ease of reading
- One or more commands inside BLOCK, separated by ;
- `my` variable inside a BLOCK will lose its value at end

Conditions II – else

- Let's warn user when we're not running BLAST
  - `else` (if the condition wasn't true…)
  - Run the code inside the else `{BLOCK}`

```
if (condition) {
    do some stuff;
}
else {     # optional
   do other stuff;
}
```

```
if ($run_blast eq "y") {
     system("blastn …");
}
else {
     print "Not running blast";
}
```

- else blocks are optional

Conditions III – else if

- See **`if_run_blast.pl`**
- Only allow "y" or "n" as inputs –
- Otherwise **`die`** (exit with an error)
- You can have one or more **`elsif`**'s after an **`if`**
  - just if, if else, if elsif, if elsif else, if elsif elsif elsif …

```
if (condition) {
    do some stuff;
}
elsif (other cond.) {     # optional
   do other stuff;
}
else {                        # optional
    do this instead;
    blocks can have >1 cmd
}
```

```
if ($run_blast eq "y") {
    system("blastn …");
}
elsif ($run_blast eq "n") {
    print "Use saved BLAST\n";
}
else {
    die "Illegal -b option\n";
}
```

Comparisons for Conditions

- String (text) comparisons: `eq ne gt lt ge le`
  - Made of letters so you know we're comparing text

```
# Compare gene names
if ($gene1 ne $gene2) {
  print "$gene1 and $gene2 are different";
}

# Careful! "y" ne "Y"
if ($run_blast eq "y") { print "Yay!\n"; }
```

- When comparing strings, "0.1" ne ".1"
  - How do we test for numerical equality?

Comparisons for Conditions II

- Numeric Comparisons: `==  !=  >  <  >=  <=`

```
if ( $num1 >= 0 ) {
    print "$num1 is positive or zero\n";
}
if (0.1 == .1) {
     print ”Oh, good. It’s a numerical comparison\n”;
}
```

- Careful!
  - `=` used to assign a variable: `$num = 37;`
  - `==` used as a test: `if ($num == 37) {…}`

- Careful!
  - Text strings have numeric value 0, so "ACTG" == "GCTA"

Multiple Comparisons

- `&&` means a logical AND (all pieces must be true)
- `||` means a logical OR (at least one piece is true)
- Group comparisons with parentheses

```
    if (($run_blast eq "y") || ($run_blast eq "Y")) {
      print "Running BLAST\n";
      system("blastn …");
    }
```

- `!` negates a condition

```
    if (!(some complicated expression)) {
         print "It wasn't true";
    }
```

Loops - foreach

- A `foreach` loop loops over a `(list)`
  - Sets a `$variable` to first value in `(list)`
  - Runs a `{BLOCK}` using that value for the `$variable`
  - Repeats loop for every value in the `(list)`
- See **`foreach.pl`**

```
foreach my $variable (list) {
   do some stuff;
   do more stuff; # …
}
```

".." is great for making lists

```
See next slide


# Given sequence $DNA of any length
foreach my $i (1 .. length($DNA)) {
  print "Letter $i of the seq is ";
  print substr($DNA, $i-1, 1),"\n";
}
```

Unrolling the loop

```
foreach my $species ("Cgla", "Klac”) {
  print "Hits for $species\n";
}
print "Hi\n"
```

```
foreach my $species ("Cgla", "Klac") {
  # Species left? Yes, "Cgla" and "Klac”. Set $species to "Cgla”
  print "Hits for Cgla\n";
} # Go back to the top of the loop, try again

foreach my $species ("Cgla", "Klac") {
  # Species left? Yes, "Klac". Set $species to "Klac"
  print "Hits for Klac\n";
} # Go back to the top of the loop, try again

foreach my $species ("Cgla", "Klac") {
  # Any species left? No. Stop looping
} # Continue the program after the loop
print "Hi\n";
```

Loops II - while

- A `while` loop keeps running while a `(condition)` is true
- It checks the `(condition)`
- Runs code in the `{BLOCK}` if it was true
- Then checks again…
- It's sort of like foreach + if

```
while (condition) {
   do some stuff;
    then do other stuff;
}
```

```
# Print numbers from 5 to 15 by fives
my $i = 5;
while ( $i < 20 ) {
    print "$i ";
    $i = $i + 5;
}
# Here, $i=20 BUT code never prints 20
# If we tested $i <= 20, we'd print 20
```

Loops III – Jumping Around

- `last` jumps out of a loop
- `next` skips to the `{BLOCK}` bottom, but then keeps looping
- Note: `if` is NOT a loop - `last` / `next` ignore `if` blocks

```
my $count = 1;
while ($count <= 10) { # repeat for up to ten species
    print "Input species $count abbreviation, or Q to end: ";
    my $species = <>;
    chomp $species;
    if ($species eq "Q") {  last; }
    elsif ($species eq "") {
        print "No species entered.\n";
        next; # no grep, counter doesn't change. Ask again.
    }
    system("grep '$species' $blast_out");
    $count = $count + 1;
}
```

Exercise – Loops and Conditions

1. Write a program to BLAST/grep four files
   - Use `"YAL001C.fasta"`, `"YAL002W.fasta"`, …
   - Hint: Add a loop to `EX_Loops_1.pl`
2. Tell user what' s happening
   - Start with solution to `EX_Loops_1.pl`
   - If file is `YAL002W.fasta`, print "It's my favorite sequence!"
3. Input checking
   - If the user inputs an e-value other than 1e-4, then using a stored BLAST output would be bad.
   - Make the program **`die`** if the user inputs -e not equal to 1e-4 and also inputs -b n
   - Hint: what compound condition do you need to test?
   - Start with `EX_Loops_3.pl`

Data Munging

Perl for Bioinformatics

or

Reading, Filtering, Merging, Changing, and Writing Data

Math

- Arithmetic operators: + - / * %

```
$a = 10 + 5; # $a is now 15
$a = $a + 20; # add 20 to the value of $a
$a += 20; # short cut, similarly -= /= *=
$a++; # shorter cut, same as $a+=1
$a = "hi" + 5; # $a=5. A text string counts as zero
```

- % is "modulus", or the remainder after division:
  11 % 3 = 2, 12 % 3 = 0

Math II - Functions

- A function takes one or more arguments
  - Math functions: sqrt, exp, log, int, abs, …
- A function returns a value
  - Set a variable equal to the return value
  - Or print it
- Parentheses are optional (sometimes)
  - Better to use them unless it's really obvious

```
$b = int(3.2); # Remove after the decimal. $b = 3
print int(-3.2); # (Or print int -3.2) prints -3
print int -3.2; # Same
```

Math III – Precedence

- Parentheses are **not** optional (sometimes)

```
$a = 4*3 + 2; # $a=14
$a = 4 * 3+2; # oops! Spaces can be dangerous
$a = 4 * (3+2); # correct. $a = 20
# quadratic equation
$x = (-$b + sqrt($b*$b - 4*$a*$c)) / (2*$a)
```

Text Functions – A Brief Overview

- `"abc" . "def"` → `"abcdef"`
- `join(":", "a", "b", "c")` → `"a:b:c"`
- `split(/:/, "a:b:c")` → `"a", "b", "c"`
- `substr("abcdefghi", 2, 5)` → `"cdefg"`
- `reverse("ACTG")` → `"GTCA" # NOT complement!`
- `"ACCTTG" =~ s/T/U/g` → `"ACCUUG" # DNA->RNA`
- `"ACCTTG" =~ tr/ACGT/UGCA/` → `"UGGAAC" # complement!`
- `length("abc")` → `3`
- `lc("ACTG")` → `"actg" # uc does the opposite`
- `index("ACT", "TTTACTGAA")` → `3 # -1 if not found`
- Wow! (**`perldoc -f split`**, etc.)

Regular Expressions

- Patterns for searching a text string
- Does the string FOO appear in variable $x?
  - `if ($x =~ m/FOO/) { print "Found FOO!" }`
  - True for $x="FOO", "aFOO", "FOOFOOFOO", "FOOLISH"
  - False for $x="", "FO", "OOF", "foo", "F O O"
  - `m/FOO/` is the same as `/FOO/`
  - `if (/blah/) {print "$_ has blah in it\n" }`
- Powerful, confusing
- **`perldoc perlretut, perlreref, perlre`**

Regular Expressions II

- `^` matches beginning of string, `$` matches end
- Many special characters must be \quoted
  - `^ $ ( ) { } [ ] ? . @ + * / \`
  - I.e., `\$` matches a literal dollar sign, not end of string
  - `\t` tab `\n` newline `\s` (space,`\t`,`\n`) `\S` non-space `\d` digit
- `/stuff/i` - the 'i' option ignores case
- See **match.pl**

```
$x =~ /ACTTGG/ # Finds subsequence ACTTGG in $x
$x =~ /^M/ # Finds seq starting with methionine
$x =~ /\*$/ # Sequence ends with stop codon
$x =~ /AACC/i # Find upper- or lower-case bases
```

Regular Expressions III

- `|` means or (sort of like `||`)
- `.` matches any character except `\n`
- `[ACT]` means any one of A, C, or T. `[A-Z]` any upper case
- `()` save (part of) a match in magic variables `$1`, `$2`, etc.
    - Can also be used to group together - see next slide
- Search for variables (another use of `$`)

```
/ACAG|ACCG/ # Matches a profile
/A.C/ # matches ABC, A1C, A C, A~C, but not AC, A\nC
if (/AC([AC])G/) { # Note: ACACG will NOT match
  print "Wobbly base was $1\n";
}
if ($line =~ /$species/) { print "Got $species!\n" }
```

Regular Expressions IV

- + matches 1 or more copies of the previous thing
- * matches 0 or more copies of the previous thing
- ? matches if something appears or if it doesn't

	/ab?c/	/ab*c/	/ab+c/
ac	✓	✓	X
abc	✓	✓	✓
abbc	X	✓	✓

Note: /ab*/ matches ac!
/^ab*$/ doesn't match ac

	/a(bc)?d/	/a(bc)*d/	/a(bc)+d/
ad	✓	✓	X
abcd	✓	✓	✓
abccd	X	X	X
abcbcd	X	✓	✓

```
/CG?CA/ # Finds sequence with or without deletion
if (/^>(\S+)/) {$id=$1} # FASTA ID (\S = non-space)
```

Substitutions

- Replace first occurrence of FOO in variable $x with BAR
  - `$x =~ s/FOO/BAR/;`
  - "aaaFOObbbFOO" → "aaaBARbbbFOO"
- Replace all occurrences
  - `$x =~ s/FOO/BAR/g; # g stands for "global"`
  - "aaaFOObbbFOO" → "aaaBARbbbBAR"
- The thing to substitute can be a regular expression
  - `$x =~ s/a+/x/;`
  - "aaaFOObbbFOO" → "xFOObbbFOO"
- Matches are "greedy" (unless you specify otherwise)
  - `$x =~ s/a.*F/x/; # non-greedy: s/a.*?F/x/`
  - "aaaFOObbbFOO" → "xOO" (non-greedy: "xOObbbFOO")
- If it can't find FOO, s/// does nothing
  - `$x =~ s/FOO/BAR/;`
  - "aaabbb" → "aaabbb"

Exercise – Regular Expressions

1. Edit `EX_Regexp_1.pl` to die unless the user inputs a valid species
   - One upper-case letter followed by three lower-case letters
2. Promise me you'll learn about regexps someday
   - **`perldoc perlretut, perlreref, perlre`**
   - "Mastering Regular Expressions" (O'Reilly)
   - Or just start using them (carefully)

I/O Overview

- Filehandle
  - A way to "hang on" to (name, refer to) a file
  - Not the same as a file name
  - Usually a name in all capital letters
- Open a filehandle to read from/write to a file
- `<FILEHANDLE>` reads a line from a file
- `print FILEHANDLE` … writes to a file
- Multiple read/write filehandles open at once
- Close filehandle when done reading/writing

Opening and Closing Files

- `open(FILEHANDLE, "filename")`
  - Must be done **before** reading/writing a file
  - Associates the file name with a filehandle
  - `"filename"` is the same as `"<filename"` - read from file
  - `">filename"` - write to file
    Note: `>` DELETES ANY PRIOR DATA IN THE FILE!
  - `">>filename"` - add to end of file. Doesn't delete anything.
  - `open(…) or die "Error: $!\n"` helps diagnose problems
- `close(FILEHANDLE)`
  - Finish writing/reading

Reading From Files

- `$x = <FILEHANDLE>;`
  - Reads from a filehandle
  - Gets one line at a time (by default)
- `<STDIN>` (abbreviated `<>`)
  - Reads from the keyboard
  - OR from files given as arguments to the script

    ```
    perl blah.pl file1 file2
    ```
  - Automatically opened/closed

I/O: Reading from a file

- Let's replace UNIX grep with Perl regexps

```
open(BLAST, "<$blast_out")
    or die "Can't open $blast_out: $!\n";
$line = <BLAST>;
if ($line =~ /\t$species/) { # species name after a tab
    print $line;
}
close(BLAST);
```

- Great, but we're only reading one line
  - Can we read multiple lines (without Repeating Code)?
  - How do we know when the file is done?

I/O: Reading from a file II

- Using a `while` loop with `<FILEHANDLE>`
  - If there are no lines left, `<FILEHANDLE>` will return `undef`,
  - `undef` is default value for variables (`my $var;`), not `""`
  - `defined($line)` is true EXCEPT if `$line` is `undef`
  - See **`read_file.pl`**

```
open(BLAST, "<$blast_out")
    or die "Can't open $blast_out: $!\n";
while (defined(my $line = <BLAST>)) {
    if ($line =~ /\t$species/) { # species name after tab
        print $line;
    }
}
close(BLAST);
```

Writing To Files

- `print FILEHANDLE "string", $var, …`
  - Prints one or more things to a filehandle
  - Remember to explicitly write `"\n"`'s
  - Note: **no comma** between FILEHANDLE and stuff to print
- `STDOUT`
  - `print STDOUT …` is the same as a regular `print …`
  - Prints to screen even if one or more filehandles are open
- See **`write_file.pl`**
- Advanced: filehandles can be variables
  - `open(my $fh, ">", "file")`
  - `print $fh "something”`
  - `while (<$input_fh>) {…}`

Parsing BLAST Output with Regexps

- lcl|Scer--YAL036C Spar--ORFN:355 92.20 1103 86 0 1 1103 1 1103 0.0 1459
- `$line =~ /^\S+\t($species\S*)\t/ or die "Bad line $line";`
- `my $id = $1;` pull out just the hit ID
- The regular expression we're searching with is:
  - `\S+` Multiple non-space chars
  - `\t` a tab
  - `($species\S*)` species name, followed possibly by non-space characters (AND parentheses save this string in `$1`)
  - `\t` tab after the ID
- `or die "…"` exit informatively if we have unexpected format
- See **`get_hit_ids.pl`**

Exercises – Input/Output and Munging

1. Write Cgla results to `Cgla_hits.txt` and Sklu results to `Sklu_hits.txt`
   - Change `EX_Munging_1.pl`
   - The easy way: read BLAST results twice
   - Slightly harder: read BLAST results only once
     - (Hint: you can have multiple input or output files open at the same time, as long as they have different filehandles)
     - Solutions are `SOL_Munging_1a.pl` and `SOL_Munging_1b.pl`
2. Edit `EX_Munging_2.pl` to also get the percent identity (next column after ID)

The Scriptome

Advanced Data Munging for Beginners or
Perl for Wimps Busy Biologists

The Default Variable $_

- Many functions act on `$_` by default
  - `print` prints `$_`
  - `chomp()` removes \n from end of `$_`
  - `while(<HANDLE>)` reads lines into `$_`
    - Same as `while(defined($_=<HANDLE>))`
    - `<>` only reads into `$_` inside a `while()`!
  - `/a/` matches against `$_` (no `=~` necessary)
  - `s/A/B/` substitutes B for A in `$_`
- If you can't find a variable, assume it's `$_`
- Give variables descriptive names

One-Liners

- Perl has shortcuts for data munging
- (You won't be tested on this!)
- fancy grep with full Perl functionality: get FASTA IDs

```
perl -wlne 'if (/^>(\S+)/) {print $1}' a.fasta > IDs
```

- sed+awk with Perl functionality

```
perl -wpe 's/^\S+\t(\w{4}--\S+).*/$1/' b.out > IDs
```

- Add line numbers to a file

```
perl -wpe 's/^/$.\t/' blah.txt > blah_lines.txt
```

- Count times each value is in col 3 (tab-separated)

```
perl -wlanF"\t" -e '$h{$F[2]}++; END { foreach (keys
%h) {print "$_\t$h{$_}"}}' blah.tab > count.tab
```

One-Liners II: Serious Data Munging

- With practice, you can *explore* your data quickly
  - Much faster than opening up a graphing program
  - Also good for "sanity checking" your results
- Choose best BLAST hit for each query sequence

```
perl -e '$name_col=0;$score_col=2; while(<>) {s/\r?\n//; @F=split
/\t/, $_; ($n, $s) = @F[$name_col, $score_col]; if (!
exists($max{$n})) {push @names, $n}; if (! exists($max{$n}) || $s
> $max{$n}) {$max{$n} = $s; $best{$n} = ()}; if ($s == $max{$n})
{$best{$n} .= "$_\n"};} for $n (@names) {print $best{$n}}' infile

> outfile
```

Scriptome Motivation

- "You can't possibly learn Perl in a day "
- "But I need to get work done!"
- "If only someone would do all the work for me…"

The Scriptome In One Slide

- Scriptome: cookbook of Perl one-liners
  - No programming needed
  - No install needed (if you have Perl)
  - No memorization needed
- sysbio.harvard.edu/csb/resources/computational/scriptome
- Read the instructions
- Find BLAST results with > 80% identity (3rd col.=2)
- Expand code to see how it's done
- Build a protocol

Sample Scriptome Manipulations

- Manipulate FASTAs
- Filter large BLAST result sets
- Merge gene lists from different experiments
- Translate IDs between different databases
- Calculate 9000 orthologs between two species of *Drosophila*

Exercises – Scriptome

1. Print BLAST hits from one_seq.blast with 80 – 85% identity (see `EX_Scriptome_1.txt`)
2. Use the Scriptome to *change* `allY.fasta`, which contains four sequences, to tabular format. (see `EX_Scriptome_2.txt`)

Arrays and Hashes

Groups of Things
for
High Throughput Munging

Why Arrays?

- What if we want to store the hit IDs?
  - Further analysis
  - Different kinds of filtering
  - Printing out
- We don't want to read the file multiple times!
- Store the IDs in an array

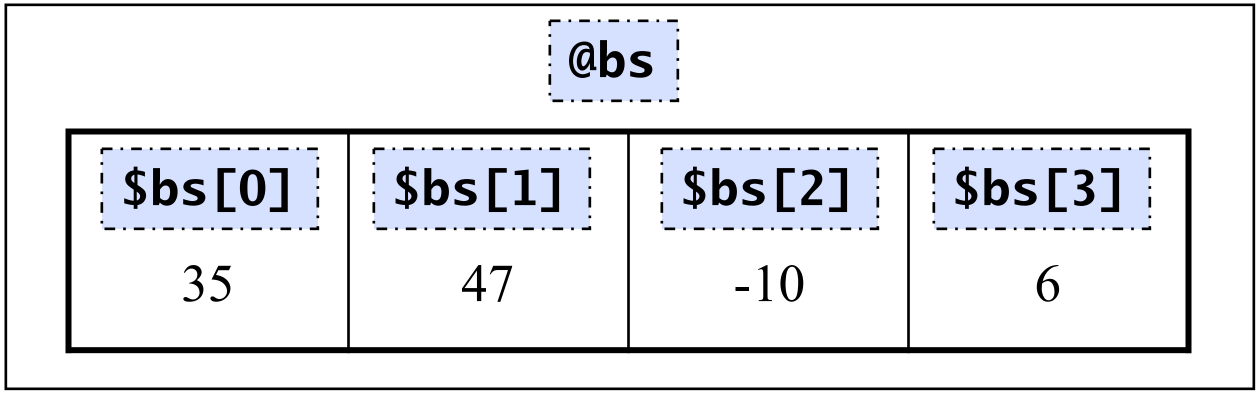
Arrays

- A box containing a **set** of "things"
  - `@bs = ( 35, 47, -10, 6 );`
  - `@strings = ("a", "b", "cde");`
  - `@scalars = ($a, $b, $c);`
- Array names start with @
- Best for many of the same kind of data
  - A set of sequences, a set of fold change values
  - Do the same thing to each array member
  - Filter to find certain useful members

@

Arrays II – Accessing the Insides

- Each thing in an array is like a scalar variable
  - So each scalar has a name that starts with $
  - It also has an index (number) to identify it
  - Indexes start from ZERO
  - `@bs = ( 35, 47, -10, 6 );`
  - `print $bs[2] # -10. Note the $`
  - `print @bs # 3547-106. Note the @`



Arrays III – Manipulating

A single value in the array can change.

```
@letters = ( "a", "b", "c", "d" );
$letters[2] = "x";
print @letters; # abxd
```

An array's size can change (unlike FORTRAN, C)

```
@nums = ( 9,8,7 );
$nums[3] = 6;
print @nums; # 9876
push @nums, 5; # push onto end - 98765
pop @nums; # pop off of the end - 9876
print scalar (@nums); # Array size = 4
```

Playing with Arrays

- `split()` splits a string into pieces
- Let's split our BLAST hits into columns
- `my @cols = split /\t/, $line;`
- Now easily access percent identity, target ID, etc.
- lcl|Scer--YAL036C Spar--ORFN:355 92.20 1103 86 0 1 1103 1 1103 0.0 1459
  - `my $percent_identity = $cols[2]; # count from 0!`
  - `print "Score: $cols[-1]\n"; # -1 is last thing in array`

  -
    ```
    # Set multiple scalars from a "slice" of an array
    my ($subj_id, $pct_ident, $align_len) = @cols[1..3];
    ```
- See **`get_hit_cols.pl`**

The Magical Array @ARGV

- `@ARGV` holds any arguments you gave your Perl script
- `perl script.pl 73 abc "Amir Karger" myfile.txt`
- `my $num = $ARGV[0]; # 73`
- `my $str = $ARGV[1]; # "abc"`
- `my $name = $ARGV[2]; # "Amir Karger"`
- `my $file = $ARGV[3]; # "myfile.txt"`
- OR `my ($num, $str, $name, $file) = @ARGV;`
- TMTOWTDI: parse `@ARGV` instead of using Getopt::Long
  - Getopt::Long will only remove –options. Files will still be in `@ARGV`
- `shift(@ARGV)` removes `$ARGV[0]`
  - `shift()` with no argument acts on `@ARGV`
  - BUT in a subroutine, `shift()` acts on `@_`

Why Hashes?

- Searching an array for a given value is slow
- Array indexes must be numbers – IDs are strings
- "A gene" has many associated pieces of data
  - Name
  - Alternate name(s)
  - Disease association(s)
  - English description
  - Coded protein(s)
- Storing diverse types of data in one array is messy
- Why can't we have arrays with string indexes?

Hashes

- A box containing a set of key/value pairs
  - Only one value per key (simple case)
- Give it a key, it returns a value
  - What NCBI ID represents "BRCA1"?
  - What amino acid does "ATG" code for?
  - What is the "DE" part of this Uniprot record? http://us.expasy.org/uniprot/Q92560
- Hash names start with %

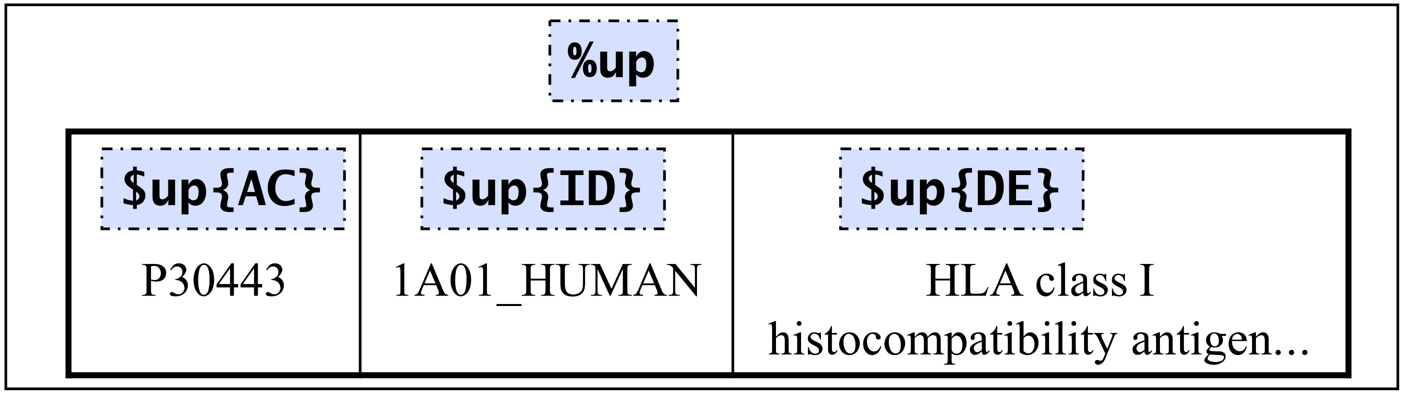
%

Hashes II - Declaration

%hash = (key1=>val1, key2=>val2, ...)

```
%up = (
  "AC" => "P30443",
  "ID" => "1A01_HUMAN",
  "DE" => "HLA class I…",
);
```

```
%translate = (
  "ATG" => "M", "GGT" => "G",
  "CAT" => "H", "TAG" => "*",
); # etc. . .
print "ATG encodes $translate{'ATG'}";
# ATG encodes M
```



Hashes III - Usage

- Accessing hashes
  - When looking at a whole hash, `%hash`
    `keys(%hash)` gets all keys in the hash
  - When accessing one value, `$hash{key}`
  - Setting one value: `$hash{key} = value;`
- Hashes vs. arrays
  - Hashes are NOT in any order
  - BUT you can get to a value instantly instead of searching through an array
  - Keys are usually text strings, not numbers
- See **`unique_hits.pl`**

Hashes IV – Common Hash Uses

- Translation table (codons, sequence IDs, etc.)
- Storing complicated records
  - Uniprot: store and manipulate ID, AC, DE separately
  - BLAST hits: manipulate ID, % identity, etc. separately
  - `my %hit = ( "ID" => $cols[1], "pct_id" => $cols[2], …);`
- See if we know about a particular thing
  - `if (! exists $known_ID{$ID}}) { do stuff…}`
- Make things unique (only one value per key)
  - Read lines into `%hash`, look at `keys(%hash)`

Exercises – Arrays and Hashes

1. Edit `EX_Array_1.pl` to print hits of any species with percent identity (third column) between 80 and 85
2. `EX_Array_2.pl` puts data from various columns (see "Hashes IV" above) into a `%hit` hash. Change the program to use that hash in the `if` and `print` statements in the `while` loop.

Perl in a Day

Class Overview

- Introduction – Why learn Perl?
- Scripting – Reproducible Science
- Variables – Making Programs Reusable
- Control Structures – Doing Things Lots of Times (Or Not)
- Data Munging – Perl for Bioinformatics
- Arrays and Hashes – Groups of Things
- The Scriptome –Data Munging for Perl Beginners
- Subroutines & Modules – Making Programs *Really* Reusable
- Objects – Complex Data, Complex Workflow
- BioPerl – Doing (More) Bio With Perl

Subroutines and Modules

Making Programs Really Reusable
by
Creating New Functions

Subroutines – Why?

```
my $dna1 = "CCGGCCGGATGTCTTAGGCGTAGCCGGCCGG"; # UTR+CDS
# (Shortest possible exon: +? is a "non-greedy" +)
$dna1 =~ /(ATG(...)+?)TAG/; # start codon, 3N bp, stop
my $len = length($1)/3; # length of translated protein

# Later…
my $dna2 = <FASTA>; # Read in DNA from FASTA file

# Do the same thing to the new sequence
$dna2 =~ /(ATG(...)+?)TAG/;
$len = length($1)/3;
```

- Harder to read larger program
- What if there’s a bug (TAG only)? Update every copy

Subroutines – Example

```
my $dna1 = "CCGGCCGGATGTCTTAGGCGTAGCCGGCCGG";
my $len = &get_translated_length($dna1); # call sub
print "DNA with UTR: $dna1. Protein length: $len\n";

my $dna2 = <FASTA>;
# Call the subroutine again, with a different argument
$len = &get_translated_length($dna2);  print $len;

sub get_translated_length {
    my ($dna) = @_; # changing $dna won't change $dna1
    $dna =~ /(ATG(...)+?)TAG/; # Remove stop codon,3' UTR
    my $plen = length($1)/3; # resulting protein length
    return $plen;
}
```

- Only one copy of the code
- Main program becomes shorter and simpler

Subroutines – View from the Outside

- Subroutines: write your own Perl functions
- *main* program *calls* subroutine
  `&get_translated_length`
  - Ampersand is optional
- It *passes* zero or more *arguments* (`$dna1`)
  - Parentheses are (sometimes) optional
- Code in the subroutine gets executed
- Subroutine *returns* results to *caller*
  - Perl subroutines can return multiple values
  - Some subroutines return no values

Subroutines – View from the Inside

```
# Comments describe the subroutine
sub some_name {                          - starts a subroutine
  # Local copies of the arguments
  my ($thing, $other) = @_;              - gets the arguments
  # Put fancy code here…                 - calculates, prints,
  # More code…                             does other stuff
  # More                                   calls other subroutines?
  return ($first, $second);              - returns stuff to caller
}                                        - ends subroutine
```

- Some people use @_ or $_[0]… in subs - careful!
- See **`sub.pl`**

Subroutines – Extra credit/FYI

- Alternate way to get the arguments inside the subroutine
  - `my $thing = shift;`
  - `shift` is like `pop`, but pulls out `$array[0]`
  - Inside a subroutine, `shift()` does `shift(@_)`
  - I.e., put the first argument to the subroutine into `$thing`

- Passing 1 array/ hash to a sub: easy. Make it the *last* arg
  - `call_sub($a, $b, @c);` Pass array to sub
  - `my ($arg_a, $arg_b, @arg_c) = @_;` Get args inside sub
- Passing 2 arrays/hashes: harder. `perldoc perlreftut`
  - `call_sub(\@arr1, \@arr2);` References "pack" arrays into scalars
  - `my ($ref1, $ref2) = @_;` Get (scalar) args inside sub
  - `@in_array1 = @$ref1;` Unpack references - scalar back into array

Subroutines – Organizing Code By Function

- Code reuse
  - Call same subroutine from different parts of your program
  - More general: `$len = &get_protein_length($dna, $remove);`
- Organization
  - E.g., separate messy math from main program flow
  - Each subroutine can only mess up its own variables
- Easier testing
  - Test subroutine code separately
- Increased efficiency
  - Write code just once, optimize just one sub
- Coder's Creed: Never write the same code twice

Modules

- A set of related subroutines
  - Placed in a separate file
  - Included in the original file with the `use` command
- We've been using modules all day
  - `use Getopt::Long;`
  - Reads in the file `/usr/…/perl5/…/Getopt/Long.pm`
  - Now `&GetOptions()` acts like a regular Perl function
  - `perldoc Getopt::Long` gets module documentation
    - Documentation is stored inside the module
    - POD, a very simple HTML-ish language
  - `strict` is a special module called a "pragma"

Modules II

- Getting new modules
  - **Thousands** of modules available at www.cpan.org
  - search.cpan.org (E.g., search for "transcription factor")
  - Usually simple to install
  - Basically, installation places .pm file(s) in `/usr/`…
  - Or a different directory Perl knows to look in
- Benefits (like subroutine benefits, but more so)
  - Organization: separate a set of functionality
  - Code reuse: don't have to re-write code for every program
    - "Good composers borrow; great composers steal." -Stravinsky?
  - Modules also give you access to new classes…

Exercise – Subroutines

- Move BLAST (and deciding whether to run) to a subroutine
- **`&maybe_run_blast($run_blast, $fasta_in, $e_value, $blast_out);`**
- Now our main program is *much* easier to read:

```
GetOptions(…);

&maybe_run_blast($run_blast, $fasta_in, $e_value, $blast_out);

foreach $species ("Cgla", "Sklu") {
    &analyze_blast($species, $blast_out, $unique_hits);
}
exit;
```

Objects and Classes

Complex Data, Complex Workflow
or
How to Write Big Perl Programs Without Going Crazy

Objects

- Scalar variables storing multiple pieces of data
  - `$uniprot_seq` stores a whole Uniprot record
  - Easier than keeping track of complicated hashes
  - Store many Uniprot records in a hash/array
- Variables that can do things (by calling *methods*)
  - `$uniprot_seq->id` gets the ID
  - Like `&id($uniprot_seq)`, but better (see below)
  - `$rev = $uniprot_seq->revcom` reverse complements

Objects II – Bio objects

- Bioperl objects store biological information
- Bioperl objects do biological things

```
use Bio::Seq;

# $seq is a Bio::Seq object, which represents a sequence
# along with associated data...
print "Raw sequence: ", $seq->seq(); # Just a regular string
print "Species is ", $seq->species();

# Object's sub-pieces can be objects too!
@features = $seq->get_SeqFeatures(); # Coding sequences, SNPs, …
foreach $feat ( @features ) {
  print $feat->primary_tag, " starts at ",$feat->start\n";
}
```

Classes

- Really just a fancy module
- Every object belongs to one or more classes
- What kind of object is it?
  - Sequence, Feature, Annotation, Tree...
- What fields will this object have?
  - species, start/end, text, subtrees
- What can I DO with this object?
  - I.e., what methods can I call?
  - id, get_SeqFeatures, set_root_node

Classes II – Bio Classes

- Bioperl classes have Bioperl objects in them, which
  - Store biological information
  - Do biological things

```
# Bio::Seq object $seq can DO things, not just hold information
use Bio::Seq;
print "Sequence from 1 to 100: ", $seq->subseq(1,100);

# You can chain -> method calls.
# revcom returns Bio::Seq object. revcom->seq returns raw sequence
$rev_comp = $seq->revcom->seq();
print "Reverse comp. from 1 to 100:", $seq->revcom->subseq(1, 100);
```

Object Oriented Programming – Who Cares?

```
# User has pulled in sequences from different databases
my @seqs = ($uniprot_seq, $EMBL_seq, $GenBank_seq);

foreach my $seq (@seqs) {
    print $seq->id;
    print $seq->description;
}
```

- Different classes can have totally different ways to implement the `id` method
- User doesn't have to care!
  - Crucial for large programs
- Each object "automagically" does the right thing
  - Because each object knows which class it belongs to
- Congratulations: you're now an OOP expert!

Bioperl

Doing (More) Bio with Perl
by
~~Stealing~~ Using Collected Wisdom

BioPerl Overview

- Modules useful for doing bioinformatics in Perl
- Many specialized modules (Annotation, Parsing, Running BLAST, Phylogenetic Trees, …)
- Many scripts
  - `which bp_seq_length.pl`
  - ``perldoc -F `which bp_seq_length.pl` ``
- Can be a bit overwhelming
  - Huge (> 800,000 lines of code, 2010)
  - Mostly uses objects
  - Documentation not always easy

BioPerl Tutorial TOC (old)

BioPerl Tutorial TOC II

Bio::Perl - Easy Bioperl

- Bio::Perl provides simple access functions.
  - Much easier than the rest of Bioperl
  - Much less functionality

• get_sequence	get a sequence from Internet databases
• read_sequence	read a sequence from a file
• read_all_sequences	read all sequences from a file
• new_sequence	make a Bio::Seq object from a string
• write_sequence	write one or more sequences to a file
• translate	translate a sequence. Return an <u>object</u>
• translate_as_string	translate a sequence. Return a <u>string</u>
• blast_sequence	BLAST a sequence *using NCBI computers*
• write_blast	write a BLAST report out to a file

Bio::Perl II - Getting Sequences

Retrieve EMBL sequence, write it out in FASTA format

```
use Bio::Perl;

# only works if you have an internet connection
$seq_object = get_sequence("embl","AI129902");

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

What could you do with `while()`? (Careful!)

Bio::Perl III - Automated BLAST

BLAST sequence at NCBI using default “nr” database

```
use Bio::Perl;

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

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

# write results to a file
write_blast(">cdna.blast",$blast_result);
```

BioPerl - Objects

- Bio::Seq: main sequence object
  - Available when sequence file is read by Bio::SeqIO
  - It has many methods - `perldoc Bio::Seq`

```
# Make a new Bio::SeqIO object $myseqs
# by opening a file for reading
#(This command doesn't actually read any sequences)
$myseqs = Bio::SeqIO->new(
  '-file' => "<inputFileName", '-format' => 'Fasta'
);

# Get next (i.e., first) seq in Bio::SeqIO object
# $seqobj is a Bio::Seq object
$seqobj = $myseqs->next_seq();
```

BioPerl - SeqIO and Seq

- Bio::SeqIO: Sequence input/output
  - Formats: Fasta, EMBL, GenBank, uniprot, PIR, GCG, …
  - Parse GenBank sequence features: CDS, SNPs, Region
  - Uses Bio::Seq objects instead of storing only sequence bp in scalar text strings
- Bio::Seq: sequence manipulation
  - subsequence
  - translation
  - reverse complement, and much more
- **`See gb2fastas.pl`**

BioPerl - SeqIO and Seq II

```
#Using SeqIO and Seq
use Bio::SeqIO;
use Bio::Seq;
$in = Bio::SeqIO->new(-file=>"<$fin", "-format"=>"Fasta");
$out =
  Bio::SeqIO->new(-file => ">$fout", "-format" => "EMBL");
while ($seq = $in->next_seq()) {
  $out->write_seq($seq); # print sequence to $out
  print "Raw sequence:", $seq->seq();
  print "Sequence from 1 to 100: ", $seq->subseq(1,100);
  print "Type of sequence: ", $seq->moltype, "\n";
  if ($type eq "dna") {
    print "Reverse comp: ", $seq->revcom->seq(), "\n";
    print "Revcom 1-100:",$seq->revcom->subseq(1, 100);
  }
}
```

BioPerl - BPlite

- BPlite: Blast Parser "lite“
  - BLAST -outfmt 6 doesn't actually give us alignments
  - But BLAST output is Hard! (see **`one_seq.long_blast`**)
  - One of several BLAST parsers available
  - Each matching sequence can have multiple matching regions ("hsp", high scoring pair)

```
use Bio::Tools::BPlite;
$report = new Bio::Tools::BPlite(-file=>"$inFile");
while(my $sbjct = $report->nextSbjct) {
  while (my $hsp = $sbjct->nextHSP) {
    print $hsp->subject->seqname;
  }
}
```

Bioperl - Codon Tables

- Bioperl::Tools::CodonTable
  - Translate/reverse translate codons & amino acids
  - Handles alternate codon tables
  - See `codon_table.pl`
  - Also includes `is_start_codon, is_ter_codon`
  - Use these codon tables to translate Bio::Seqs

What's missing

- More Bioperl, regexps, functions, OOP, ...
- Testing, debugging and proactive error checking
- Context and other shortcuts
  - $line = <FILE> reads just one line
  - @foo = <FILE> reads an entire file into an array
- Databases and web programming
- Graphics
- Perl Golf and Obfuscated Perl
  - `perl -le '$_*=$`%9e9,//for+1=~/0*$/..pop;print$`%10' 10`
- Etc.

Resources for After the Class

- **`amir_karger@hms.harvard.edu`**
- perldoc perl (see "Tutorials" section)
  - perlintro, perltut, perlfunc, perlretut, perlboot
- http://bip.weizmann.ac.il/course/prog/
  - HUNDREDS of slides - many bio-related examples
  - Also look at "assignments" for practice
- Books
  - Beginning Perl for Bioinformatics is designed for biologists. (It has a sequel, too.)
  - *Learning Perl* is more general, but gets rave reviews

Resources for After the Class II

- search.cpan.org
  - 114,000 modules and counting
- http://www.bioperl.org
  - "Howtos" Sequence Analysis, Phylogenetics, etc. using Bioperl. Lots of stealable sample code
  - bioperl-l@bioperl.org - ask questions to experts.
- http://www.nostarchpress.com/perloneliners
  - A whole book of Perl one-liners, with explanations
- The Scriptome
  - http://sysbio.harvard.edu/csb/resources/computational/scriptome

Survey

- http://hmsrc.me/introperl2016-survey1
- Please fill out the survey so we can improve our classes