Using perl for Bioinformatics

Overview

- Starting perl and creating perl programs
- Variables
- Subroutines

Basic `emacs` usage

Starting `emacs`

Start emacs with the commands

```
athena% add seven
athena% bemacs &
```

Normally, it's called `emacs`, but I've made a wrapper script for it so that you don't have to worry about setting environment variables, and so on.

Opening and editing files in emacs

Type `C-x C-f ~/.environment<RET>`. That is: Hold down the control key, and press and release `x`, then `f`. Release the control key, then type `~/.environment`, and press return.

Files in emacs, continued

Add this line to the text in the resulting window

```
add seven
PERL5LIB=/mit/seven/lib/site_perl/5.6.0
```

Saving files

To save the file, type `C-x C-s`. That is: Hold down the control key, and press and release `x`, then `s`.

The `~/.environment` file

Check that you modified the file correctly by opening up another `xterm`, and typing

```
athena% source ~/.environment && which pw
```

If this doesn't result in any obvious error messages, it shouldn't be necessary to type "`add seven`" next time you log in.

Customization for non-Athena machines

If you have trouble working with perl on Athena machines, let me know, and I'll straighten things out. If you want to set up your personal machine for 7.91, I'm afraid you're on your own. I understand how desirable such an arrangment can be, though, so here are a couple of pointers. It's going to be a fair amount of work, though.

It's probably not worth it to try to set things up on Windows machines. You could try installing `ActivePerl` from `http://www.activestate.com/Products/ActivePerl/` and installing Bioperl by hand from there, but I have no idea whether that'll work or not.

For Unix machines, you need at a perl version later than 5.6. Install the packages in `/mit/seven/src/bioperl/`

E.g.

```
tar zxf bioperl-1.2.tar.gz
cd bioperl-1.2
perl Makefile.PL
make install
```

There are some more Bioperl installation notes in
`/mit/seven/src/bioperl/README`

To get the emacs enhancements, put the code in `/mit/seven/7.91/dotfiles/emacs` in your `~/.emacs` file.

More information about `emacs`

You're going to be using `emacs` a lot. It's best you get comfortable with it as quickly as possible.

Documentation commands

`C-h t`	`emacs` tutorial (highly recommended.)
`C-h i Info<RET>`	Manual for `emacs` documentation system.
`C-h i Emacs<RET>`	Manual for emacs.
`C-h ?`	All help commands.

Editor commands

`C-g`	Abort.
`C-x u`	Undo.

First steps in perl programming

In your terminal, make a directory for your perl programs like so:

```
athena% mkdir ~/7.91
```

In `emacs`, open up the file `~/7.91/hello.pl`, and put this in it:

```
use strict;
print "hello, world!\n";
```

Now, at the terminal, type this:

```
athena% pw -w hello.pl
```

The resulting output will be “`hello, world!`”.

Things to note.

- *Always* begin your programs with "`use strict;`". It will save you a lot of grief, later.

- The `pw` command is shorthand for

  ```
  /mit/perl5/bin/perl -w
  ```

  Always use the `perl` in the `perl5` locker. It has much more functionality than the local one. If you develop on some other platform, *always* pass the `-w` switch to `perl`

- All commands end with semicolons.

Documentation

Put the cursor on the word "`print`", and type `C-c C-h f`. You will get the documentation for the `print` command. Try it on the word "`use`," too.

Put the cursor on the word "`strict`", and type `C-c C-h m`. You will get the documentation for the `strict` module.

The latter keybinding is the most reliable, but the former produces documentation in info format, which can be helpful.

Don't forget google: searching for "`site:bioperl.org BLAST`" returns pointers to documentation of bioperl's `BLAST` functionality. Searching for "`perl list scalar context`" returns pointers to explanations of how functions in perl can return different values depending on the context in which they're called.

Variables

You can use the debugger to play with perl expressions like so:

```
athena% perl -d -e 0
main::(-e:1):   0
  DB<1> $a = 1
  DB<2> print $a
1
  DB<3> $a = "foo"
  DB<4> print $a
foo
  DB<5> print "interpolation of $a"
interpolation of foo
```

Variables starting with “$” are called *scalars*.

Some bioinformatics

Create a file `bptranslate.pl` containing the following:

```
use strict;
use Bio::Perl qw(read_sequence);
my $seq = read_sequence(shift @ARGV);
print $seq->translate()->seq(), "\n";
```

This takes a nucleotide sequence file, tries to guess the file format from its extension, and prints out the standard translation of the first sequence in the file. Use it like so:

```
athena% cd /mit/seven/7.91
athena$ pw bptranslate.pl control.fa
ALRLPIKSLISCVFVCRLRYI*DSCSPWWPKTPTPPG...
```

Things to note

- You need to declare the variables you use with "`my`". This is due to the "`use strict;`" command. Without the `strict` module, variables that have not been seen before are initially assigned a default trivial value, which can get very confusing if you typo a variable name.

- The script gets the filename passed on the command line with the command "`shift @ARGV`". The variable `@ARGV` is an *array*, and `shift` returns the first element and removes it from the array.

- After printing the translation, we ask it to print "`\n`", which is the symbol for a newline. Otherwise, the subsequent `athena` prompt shows up on the same line.

bioperl objects

- You access variables within modules using “::”.

- The `read_sequence` function returns a `Bio::Seq` *object*, which we assign to the variable `$seq`. This object has a *method*, `translate`, a function which returns another `Bio::Seq` object containing the translation to protein. This object is converted into an actual *string* (sequence of characters) using the `seq` method.

- You can read about the `Bio::Seq` module by putting your cursor on it, and typing `C-c C-h m`, or by typing `perldoc Bio::Seq` in your terminal window.

Our own translator

In `/mit/seven/7.91/perl_module/translate.pl`, there is a translation program that does not depend on Bioperl. I'll go through it because it introduces some important perl concepts.

```
sub translate{
    my $sequence = shift;
    $sequence = uc($sequence);
    my $seqidx; my $codon; my @codon_list;
    for ($seqidx = 0;
         $seqidx < length $sequence ;
         $seqidx += 3) {
      $codon = substr($sequence, $seqidx, 3);
      if (length $codon == 3) {
        push(@codon_list, $codons{$codon}||'X');
      }
    }
    return ( join ('' , @codon_list));
}
```

Running the debugger

Open up `translate.pl` and press `C-c C-c` to start the debugger. Enter "`s`" twice, then keep entering "`n`" to get a feel for how the `translate` subroutine works.

The "`s`" command "steps into" the context of the function that is about to be called. That's how we get the debugger into `translate`. The "`n`" command "steps over" the command that is about to be executed.

You can evaluate expressions in the current context using the "`x`" command:

```
  DB<1> x @codon_list
  empty array
  DB<2> x $seqidx
0  0
```

Exercise

If you have time, it'll be very instructive to use the debugger to step through the calls to `read_sequence` and `seq->translate` in `bptranslate.pl`. (If you see anything interesting, try looking it up using `C-c C-h m`.)

Things to note about `translate.pl`

- The `%codons` variable is a *hash*: a mapping between arbitrary key-value pairs. In this case, it maps the codons to their respective residue symbols.

- The block under `translate` is a *subroutine*: a piece of code that you can call repeatedly, with different *arguments*. If you *call* it like “`translate("acgactagcaattcaca");`, it gets *passed* one argument: the string `"acgactagcaattcaca"`. Within `translate`, the argument is assigned to `$sequence` using the `shift` command.

More on `translate.pl`

- The `uc` command converts `$sequence` to uppercase.

- The `for` block causes `$seqidx` to iterate over the values `0, 3, 6, 9, ....` The function `substr($sequence, $seqidx, 3)` returns the substring of length three starting at each of these positions.

- Triplets of nucleotides are translated by looking them up in the `%codon_list` hash.

- The `@codons` list stores the translated residues. They get added to the end of the list with the `push` command.

- The list of residues in `%codon_list` are joined together into a string using the `join` command.