# An Introduction to Perl Programming

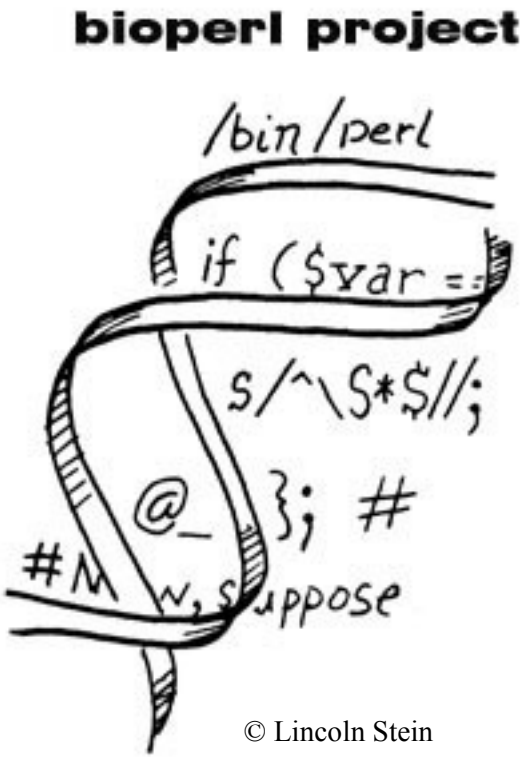
Fernán Agüero
Instituto de Investigaciones Biotecnológicas, UNSAM
fernan@unsam.edu.ar

# What is Perl

- Perl is a programming language
- Perl, as a programming language, is
  - **Interpreted**
    - You don't need to *compile* your code
  - **High-level**
    - You don't need to write 1s and 0s to make the computer understand you
  - **Dynamic**
    - The language can be modified at runtime
- Created by Larry Wall
  - **Perl is a language to get your job done!**
- In Perl
  - **There is more than one way to do it**

# Why Perl

- Perl has been designed with text processing in mind
  - Filter text, generate reports
- Ideally suited for sequence analysis
  - All sequences are text
  - Convert formats easily with Perl
    - GenBank
    - FASTA
    - Clustalw
  - Analyze and process files
    - Find restriction enzyme sites
    - Motifs
    - Vector
    - Trim sequences
    - Etc.



© Lincoln Stein

# A basic Perl program/script

- A Perl Program is
  - A plain text file
  - Containing statements written in the Perl language
- Any text editor can be used
  - Vi, vim, emacs, xemacs, nedit, gedit, pico, nano, ee
  - Textpad, PSPad (Windows)
  - BBEdit, TextWrangler (Mac OS X)
  - Remember: a Word Processor IS NOT a text editor
- In Unix a text file can be executed
  - Giving execution privileges: '`chmod +x program.pl`'
  - Using the *Shebang* mechanism
    - The first line of the text file starts with
      - `#!` (the she-bang)
    - Followed by the PATH of the program that should interpret and understand the rest of the file (statements)
    - '#!/usr/bin/perl'

# A basic Perl script

```
#!/usr/bin/perl

statement;
statement;

one long
statement
using many lines;

exit; # optional
```

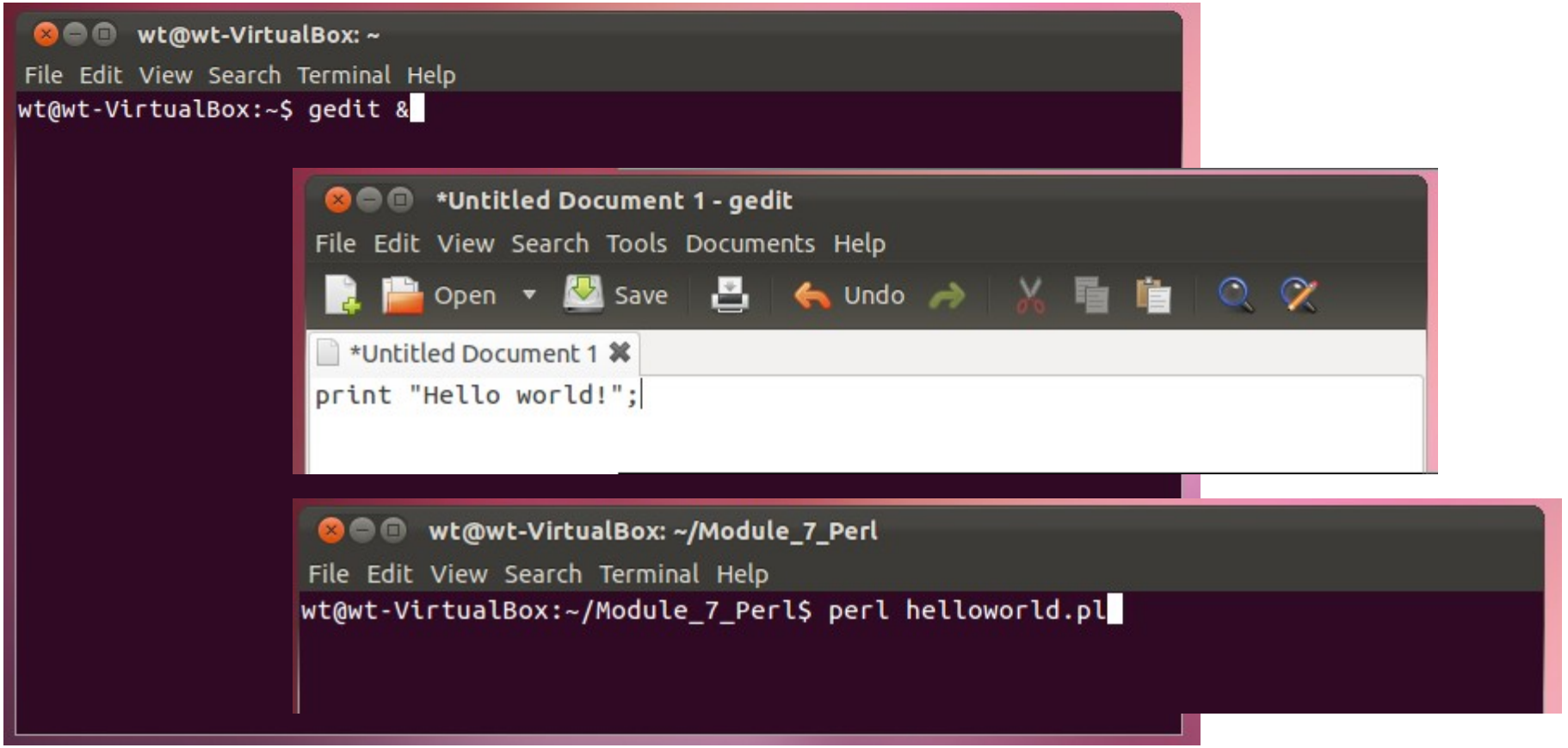
Absolute PATH to the Perl interpreter
/usr/local/bin/perl ; /opt/bin/perl

Statements are ended by semicolons

**which perl**
**whereis perl**
will give you the PATH to your Perl

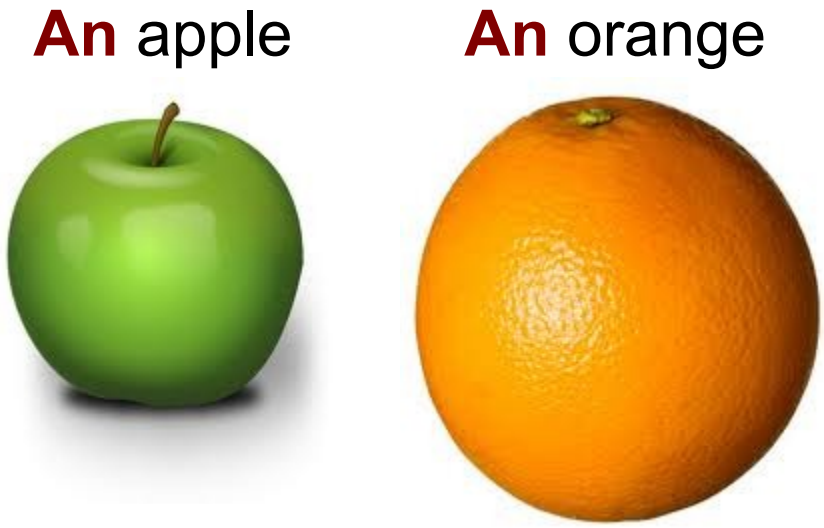
# Hello World!

## Open an editor (gedit), type, save, run ...



# Variables

- Scalars ($)
  - Unidimensional
  - Can hold any type of data
    - Text
    - Integers
    - Floating point numbers
  - They are prefixed with $




```
$var = "GGATCCGGGACCAAAA"; # assign a string

$val = 42; # assign a number

($a, $b, $c) = ("me", "my", "mine"); # assign all at once

($l, $r) = ($r, $l); # swap values
```

# Variables (cont'd)

## •Arrays (or Lists) (@)

- In Perl, an array/list is an indexed collection of values.
- Values can be scalar values of any type
  - text, numbers
- The first index starts at position 0 (zero).
- They are prefixed with @

```
@list = (“juan”, “jose”, “fred”); # assign 3 elements

print $list[0]; # print first element of $list

push @list, “roberto”; # adds string at the end

print $list[3]; # prints “roberto”

$first = shift @list; # get leftmost value

$last = pop @list; # get rightmost value
```

# Variables (contd)

- Hashes (%)
  - Also called associative arrays
  - They store values in pairs
    - Key => Value
  - They are prefixed with %

```
%me = (
  name => “Fernan”,
  age => 37,
  loves => “Perl”,
); # create a hash with 3 key/value pairs

print $me{name}; # print value associated with 'name'

$me{born} = “Buenos Aires”; # add a new key-value pair
```

# Using variables

- Choice of variable types gives you power
  - Use the variable type that best fits your data
- Getting complex
  - You can create more complex data structures by mixing scalars, arrays and hashes.
  - Some examples:
    - A hash of hashes to store sequences

```
%sequences = (
  eco0001 => { seq => "ATG...TGA", desc => "hypothetical protein" },
  eco0002 => { seq => "ATG...TAA", desc => "DNA polymerase" },
  eco0003 => { … }
);
```

# From strings to lists and back again

- Convert a string into a list of values
  - Useful when reading files exported from spreadsheets
  - E.g. from Excel, in tab- or comma-delimited format
  - @values = split( /pattern/, $string )

```
$string = "Cel1980.1 ATG Hypothetical 2.54 High";

@values = split(/ /, $string);

# @values are now ("Cel1980.1", "ATG", "Hypothetical", "2.54", "High")
print $values[3]; # would print 2.54

($id, @values) = split(/ /, $string);
# $id is "Cel1980.1", @values is now ("ATG", "Hypothetical", "2.54", "High"
```

Convert a list of values into a string

Useful to generate files that can then be imported into a spreadsheet application (OpenOffice, Excel)

$string = join( “character(s)”, @list )

```
@list = (“Cel1980.1”, “ATG”, “Hypothetical”, “2.54”, “High”);

$string = join(“||”, @list);

# $string is now “Cel1980.1||ATG||Hypothetical||2.54||High”
```

# Working with files

- Declare a handle and associate it with a file
- Use the handle to refer to the file
  - **Reading from files:**

```
open(MYHANDLE, "/home/fernan/somefile.txt");

# read the first line
$line1 = <MYHANDLE>;

# read the second line
$line2 = <MYHANDLE>;

# read all lines
while ( $line = <MYHANDLE> ) {
 # read the file one line at a time, do some action on $line
}
close MYHANDLE;
```

## Writing to files

### This will overwrite the contents of the file!

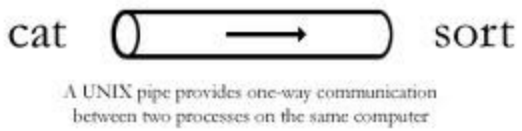
```
open(MYOUTPUTHANDLE, ">/home/fernan/somefile.txt");
print MYOUTPUTHANDLE "Hello there!";
close MYOUTPUTHANDLE;
```

### But this will just **append** the contents to the end of the file ...

```
open(MYOUTPUTHANDLE, ">>/home/fernan/seq.fasta");
# print out a sequence in FASTA format
print MYOUTPUTHANDLE ">mysequence";
print MYOUTPUTHANDLE "ACGT...";
close MYOUTPUTHANDLE;
```

# Working with files (contd)

- Special handles
  - They are always open, and available
  - STDIN, for reading (i.e. from pipes)
  - STDOUT, for writing
  - STDERR, for writing



A UNIX pipe provides one-way communication between two processes on the same computer

```
# perl myprogram.pl < input.tab
# cat input.tab | myprogram.pl
while (<STDIN>) { # read from the keyboard or from a pipe }


# perl myprogram.pl > program_output.txt
print STDOUT "MW: 325.08 kDa", "\n", "pI: 9.54", "\n", "Length: 2954 aa";

print STDERR "Warning: sequence length is zero!";
```

# Operators

- Assignment operators
  - = += .=

```
$a = 1;
$a += 2; # $a is now 3
$a *= 2; # $a is now 6
```

```
$a = "Me";
$a .= "Myself"; # $a is now "MeMyself"
$a .= "AndIrene"; # $a is now
"MeMyselfAndIrene"
```

- Control operators
  - && || ! logical AND, OR and NOT

```
if ( $a && $b ) { # do something }
if ( $mw > 100 || $pi < 9 ) { ... }
if ( ! defined $c ) { ... }
```

- Comparison operators
  - Numerical < > <= >= != == <=>
  - String lt gt le ge ne eq cmp

```
if ( $a == 4 ) { # do something }
if ( $b eq "ATG" ) { ... }
```

# Iterations, Loops

- while (expression) { execute block }
- unless (expression) { execute block }
- do { execute block } until ( expression )
- foreach (@list) { execute block }
- for (initial; expression; increment) { execute block }
  - for ($i = 0; $i >= 100; $i = $i + 1)
    - Start at zero (0),
    - Continue while $i <= 100,
    - Increment $i by one each time
- Execute block
  - List of statements that will be executed in the loop or if the condition is met
- Expression
  - An expression that evaluates to either true or false

# BioPerl

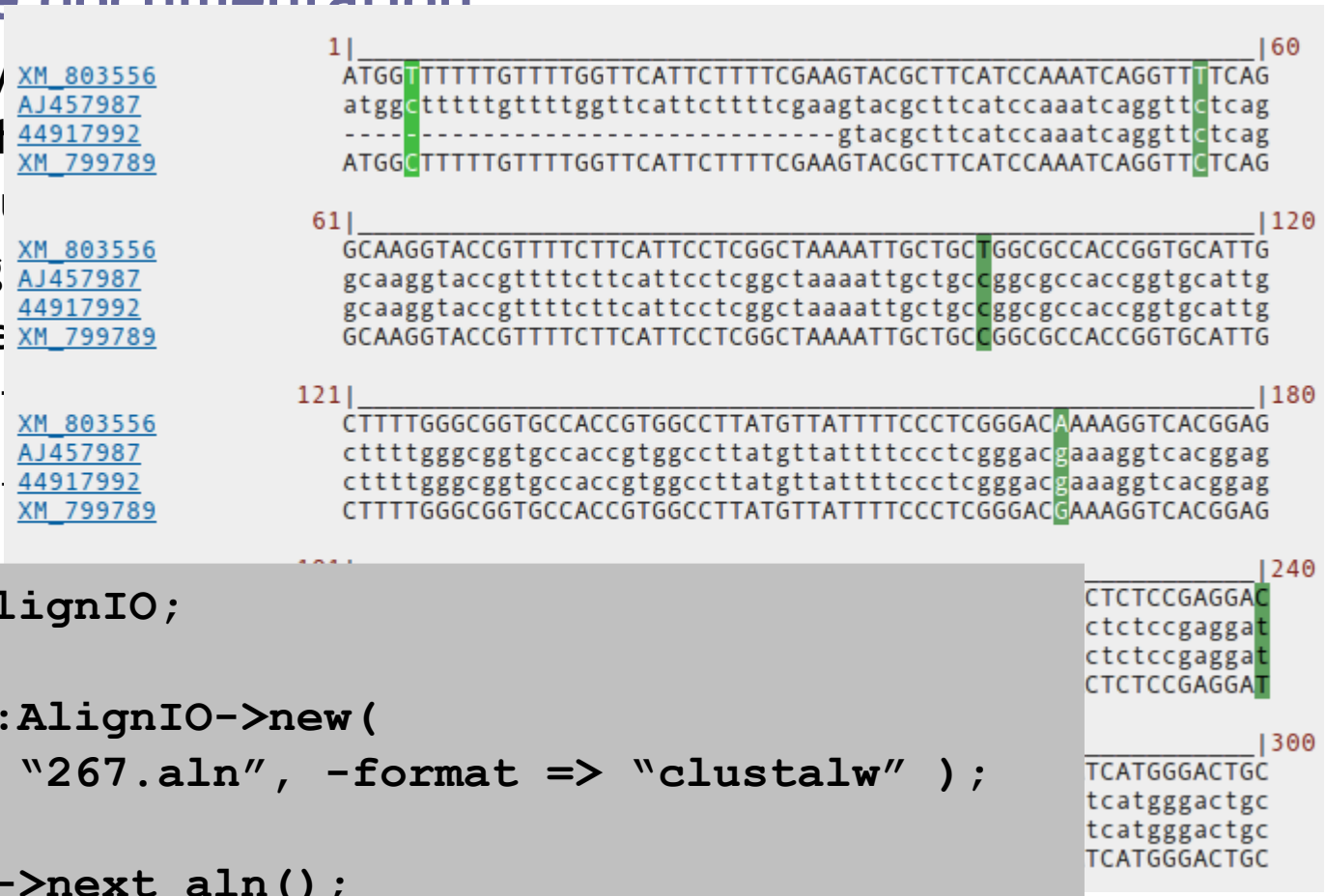
- BioPerl is
  - A collection of Perl modules
  - That greatly sim
- BioPerl allows
  - You don't need
    - Reads FASTA

```
>Tc00.1047053510665.4
ATGCCACAAGGTGGACGGGGTCCCATGGTGCTGCTAAAGCAGCAGAGGTTGCCCGCATGG
CAGCCCATCCTTACACCTCCACATGTTGCGCTAGCTTTTTTCCTGCTGAGCATATTGTTT
ATCCCTTTAGGGGTTTTCGTGACGCTAATGAACAAACAGGCGAAGGAGGTCACCGTTCGT
TATGATCATATTCATCGCTGCACAATTACACATAACACAGGCGCCTTTATGTATGAAGGC
AACAATATGACGTTTAAAACTGGATGTATGACGGAAGTTTCCTTTGATATCACGGAGAAA
CTTAAGGCCCCCGTATATCTTTATTATGAATTAACAAGGTTTTATCAAAATCACAGGCGG
TATTCAATCTCGCGAAACGATGAACAATTGGCCGGTAAAGCCGTGAGATATTTGCCAGAC
ACGTCACCTCTTACCATACCAGGGGATATTTATGGAATCTCTGGGACTCCCATCAAATAC
GTGGATGGTTCGGATTTGCGTTACAAGGATTTTTTGTACGTTCCAGCCGGCCTCATTGCC
                                                ATGGTGGAACTCCA
                                                TTCCTCTTAATGGC
                                                CGGACGTGGAATAT
                                                CGGCAAAAGAATTG
                                                TCAACAAGGGGTGG
                                                ATTTTATGGTTTGG


                                                TTTTTGTTTACTGT
                                                TCGGTTTTGCTTTG
                                                AGGCCATAAATGAA
                                                ATCGAATTTTTGAG
                                                AGGAACATATTACA
                                                GTATTAGAAAAATG
                                                GCGCTCTTGTTAAT
                                                GCGATATTGGAAGT
                                                AAAAAGTGGTGGGA
                                                CAAATGCCAAGCTT
                                                ATGATTTT

                                                TAAGGCGCACGCTG
                                                ATTCCATGGGCCTC
                                                GGCAAATGGATATT
                                                TGAGCTTCATTGTG
```

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

while ( $seqobj = $seqio->next() ) {
  $sequence = $seqobj->seq();
  # $sequence is now "ATGCCACAAGG..."
  $id = $seqobj->id();
  # $id is now "Tc00.1047053510665.4"
}
```

# Using BioPerl

- Read the documentation
  - Identify
  - Know t
    - Seq
    - Alig
  - And the
    - nex
    - add




```
use Bio::AlignIO;

$io = Bio::AlignIO->new(
  -file => "267.aln", -format => "clustalw" );

$aln = $io->next_aln();
$minialn = $aln->slice(20,30);
$newaln = $aln->remove_columns(['mismatch']);
```

- Fast conversion of formats

```
use Bio::AlignIO;

$in = Bio::AlignIO->new( -file => '1.aln”, -format => “clustalw”);
$out = Bio::AlignIO->new( -file => 2.pfam”, -format => “pfam”);

$aln = $in->next_aln;
$out->write_aln($aln);
```

```
use Bio::SeqIO;

$in = Bio::SeqIO->new( -file => '1.gbk”, -format => “genbank”);
$out = Bio::SeqIO->new( -file => 2.fasta”, -format => “fasta”);

$seq = $in->next_seq;
$out->write_seq($seq);
```

# Regular Expressions

**"...a regular expression provides a concise and flexible means for "matching" (specifying and recognizing) strings of text, such as particular characters, words, or patterns of characters."**

**Examples:**

the sequence of characters "car" appearing consecutively in any context, such as in "car", "**car**toon", or "bi**car**bonate"

the sequence of characters "car" occurring in that order with other characters between them, such as in "I**c**el**a**nde**r**" or "**c**h**a**ndle**r**"

the word "car" when it appears as an isolated word ([space]car[space])

the word "car" when preceded by the word "blue" or "red"

the word "car" when not preceded by the word "motor"

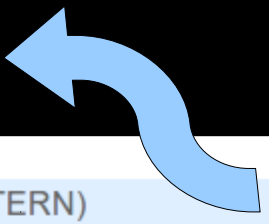
a dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits (for example, "$100" or "$245.99").

http://en.wikipedia.org/wiki/Regular_expression

# Regular expressions (cont'd)

## In Perl:

```
if ( $var =~ /GGATCC/ ) { # do something }
if ( $var =~ /A+/ { # a polyA }
if ( $var =~ /[TC]+/ ) { # a polypyrimidine tract }


if ( $aa =~ /[LIVMF](2)DEAD[RKEN].[LIVMFYGSTN]/i)
 { # a DEAD-box helicase }
```



DEAD_ATP_HELICASE, PS00039; **DEAD-box subfamily ATP-dependent helicases signature** (PATTERN)

*Consensus pattern:* **[LIVMF](2)-D-E-A-D-[RKEN]-x-[LIVMFYGSTN]**

*Sequences known to belong to this class detected by the pattern:* ALL, except for YHR169w

*Other sequence(s) detected in Swiss-Prot:* 14.

- Retrieve an alignment of Swiss-Prot true positive hits:
  Clustal format, color, condensed view / Clustal format, color / Clustal format, plain text / Fasta format
- Retrieve the sequence logo from the alignment
- Taxonomic tree view of all Swiss-Prot/TrEMBL entries matching PS00039
- Retrieve a list of all Swiss-Prot/TrEMBL entries matching PS00039
- Scan Swiss-Prot/TrEMBL entries against PS00039
- view ligand binding statistics

# CPAN

- CPAN is the Comprehensive Perl Archive Network
  - **Your one stop shop for everything Perl**
    - Modules, Frameworks
  - http://search.cpan.org

# Getting help

- Read the Perl Manual
  - `'man perl'` (Overview and links to other parts of the manual)
  - `'man perlfunc'` (Perl Built-in functions, i.e. split, join, chomp)
  - `'man perlop'` (Perl Built-in operators, i.e. `+ && < =>` )
  - `'man perlre'` (Perl regular expressions)
- Perldoc
  - `'perldoc -f chomp'`
  - `'perldoc -f print'`
  - `'perldoc -f sprintf'`
    - Get documentation for one of Perl's built-in functions
  - This also include external and/or third-party modules
  - `'perldoc Bio::AlignIO'`
  - `'perldoc Bio::SeqIO'`

# Further Reading

- Mastering Perl for Bioinformatics
  - James Tisdall
  - O'Reilly and Associates, 2003
- Learning Perl
  - Randal Schwartz, Tom Phoenix, Brian D Foy
  - O'Reilly and Associates, 5th Edition, 2008
- Intermediate Perl
  - Randal Schwartz, Brian D Foy, with Tom Phoenix
  - O'Reilly Media, 2006

# Exercises

- Getting started with Perl
  - Reformat the output of InterPro
- Generate input for Artemis
  - Read the output from a gene prediction program
  - Generate a .tab file (feature table format) for Artemis

# Exercises

- Read the following program, and try to explain its purpose, and expected input and output

```
 1 #!/usr/bin/perl
 2
 3 # In Perl $/ is a special variable
 4 # $/ is the record separator or $RS
 5 # by default $/ is a newline, we redefine it here
 5 $/ = '>';
 6
 7 while ( $next = <STDIN> ) {
 8   chomp;
 9   next if not defined $next;
10   ($header,@seq) = split(/\n/, $next);
11   $seq       = join('', @seq);
12   $seq       =~ s/\s//g;
13   $seq       =~ s/(.{60})/$1\n/g;
14   print ">$header\n$seq\n";
15 }
```