Perl Intro for Bioinformatics

Eric Mulvaney
University of Windsor

Why Perl?

- Perl is a very, very concise language.
- Perl comes with extensive documentation.
  - See the *perl* and *perldoc* manpages for more.
- The Perl compiler and runtime system can provide an amazing amount of help to the user.
  - In particular, see *perldoc diagnostics*.
- Perl is free, tried and true, and available for all major operating systems.
- Bioperl (http://www.bioperl.org/)

Basic Datatypes

- Perl has three main datatypes:
  - Scalars
    - Scalars hold single values, like integers, floating-point or real values, and strings.
  - Arrays
    - Arrays hold any number of scalars.
  - Hashes
    - Hashes associate keys with values. Both keys and values are scalar.

Scalar Variables

- All scalar variables begin with dollar-signs.

```
my $name = "Tim";
my ($dx, $dy) = (10, 20);
my $dist = sqrt ($dx**2 + $dy**2);
```

- Perl needs to be told to check for declarations.

```
print $x + 2;          # Assumes $x = 0, and prints 2
use strict;
print $x + 2;          # Error, $x undeclared!
```

Dynamic Typing

- Scalars can be integers, floats, or strings.

```
my $id = 123_456_789;
print "ID = ", $id;               # prints ID = 123456789
$id = "123 456 789";
print "ID = ", $id;               # prints ID = 123 456 789
```

- Perl will automatically convert, but be careful.

```
$next_id = $id + 1;
print "ID = ", $next_id;          # prints ID = 124
```

Strings

- Double quotes interpolate values.

```
my $name = "Tim";
print "Hello, $name!";          # prints Hello, Tim!
print 'Hello, $name!';          # prints Hello, $name!
```

- You concatenate strings with '.' not '+'.

```
print 'Hello, ' . $name . '!';  # prints Hello, Tim!
print '4' . '2';                # prints 42
print '4' + '2';                # prints 6
print 2 + 2 . 2                 # prints 42
```

More Strings

- You can extract substrings from strings.

```
my $msg = "I am a fish!";
print substr($msg, 7, 4);       # prints fish
```

- Or replace them with new substrings.

```
substr($msg, 7, 4) = "hologram";
print $msg;                  # prints I am a hologram!
```

- But you can't subscript strings like arrays.

```
print $word[0];              # Error: doesn't mean what you
                             # think it means.
```

Other String Operators

- You can duplicate strings with 'x'.

```
print "Fish" x 3;                  # prints FishFishFish
```

- Like numbers, strings can also be compared.

```
# Numbers      <   <=     == !=     >= >      <=>
# Strings      lt  le     eq  ne    ge  gt    cmp
```

- Remember, Perl does automatic conversions.

```
print '42' < '6'  ? 'true' : 'false';      # prints false
print 42 lt 6     ? 'true' : 'false';      # prints true
```

Arrays

- Arrays are like strings of scalar values.

```
my @stuff = (3.14, 42, "hands of blue");
my ($pi, $ans, $scary) = @stuff;
```

- Indexing returns a scalar, hence the '$'.

```
print "Two by two, $stuff[2]!";
```

- You can also extract slices, replace slices, but you cannot nest arrays within other arrays.

```
my @slice = @stuff[0..1];      # (3.14, 42)
@slice[1..1] = (2, 3);         # (3.14, 2, 3)
my @more = (@slice, 5, 7);     # (3.14, 2, 3, 5, 7)
```

Arrays and Strings

- Strings can be split, arrays can be joined.

```
my @girls = split(" ", "Zoe Inara Kaylee River");
print join(", ", @girls);           # prints Zoe, Inara, Kaylee, ...
print join(", ", sort @girls);    # prints Inara, Kaylee, ...
```

- Array length and string length are different.

```
print length($girls[1]);     # prints 6
my $len = @girls;
print $len;                        # prints 4
print @girls;                     # prints InaraKayleeRiverZoe
print scalar @girls;           # prints 4
```

Hashes

- Hashes are like Dictionaries in Java.

```
my %ages = ('Tim' => 42, 'River' => 17, 'Summer' => 21);
print $ages{'River'};      # prints 17
```

- Keys may not be kept in the order supplied.

```
print join(", ", keys %ages);      # prints River, Tim, ...
print join(", ", values %ages);    # prints 17, 42, 21
```

- Keys can be unquoted if they're barewords.

```
delete $ages{Tim};
print exists $ages{Tim} ? 'true' : 'false';   # prints false
```

Advanced Datatypes

- In Perl, arrays and hashes can only hold scalars, but there is another kind of scalar: references.
- References are like pointers.

```
my @array = (1, 2, 3, 4);
my $aref = \@array;
```

- Dereferencing can get messy.

```
print @{$aref};          # prints 1234
print ${$aref}[0];       # prints 1
print $$aref[1];         # prints 2
print $aref->[2];        # prints 3
```

More References

- There are easier ways to create references.

```
my $href = { Numbers => [ 1, 2, 3, 4 ] };
print @{$href->{Numbers}};    prints 1234
```

# Without the @{}, Perl will only print the pointer value.

```
print $href->{Numbers}->[0];    prints 1;
print $href->{Numbers}[1];      prints 2;
```

# Only the first -> is mandatory; without it, Perl would assume you were looking up 'Numbers' in %href.

- You can even make references to literals.

```
my $ten = \10;
$$ten = 12;           # Error: read-only value.
```

Data Structures

- Refs enable us to create complex data structures.

```
my %people = (
        River => { age => 17, siblings => ['Simon'] },
        Summer => { age => 21, gender => 'female' }
);
print $people{River}{age};                # prints 17
     # Perl figures out that $people{River} is a reference.
print @{$people{River}{siblings}};    # prints Simon
     # Since we want the whole array, we have to use @{}
```

- A good tutorial on Perl references can be found in the *perlref* manpage.

Conditionals

- Perl has the traditional if-statement, but it also has 'unless'; note, the braces are not optional.

```
my $x = $value <=> 42;
if( $x < 0 )      { print "Too small!";    }
elsif( $x > 0 )   { print "Too large!";    }
else              { print "Just right!";   }
unless( @work )    { print "Done."; }
```

- You can even suffix them for single statements – no braces, and the parentheses are optional.

```
die("Can't open $file!") unless open(IN, "< $file");
```

Loops

- It also has for-, while-, and also foreach-loops.

```
for(my $i = 0; $i < @items; ++$i)    { print $items[$i]; }
foreach my $item (@items)            { print $item; }
while( not $done ) { ...do something... }
do { ...something... } until( $done );
```

- Like if/unless, you can suffix each except for(;;).

```
print "I am a fish!" while 1;      # Infinite loop.
print $_ foreach @items;           # Can't name the iterator.
```

- Perl uses next/last like C/Java's continue/break.

```
foreach my $x (@a) { next if $x < 1; ... }
```

File I/O

- Arguments to 'open' resemble Unix sh redirects.

```
open IN, "< $file";       # IN is the file handle.
open OUT, ">> $log";      # Append to $log.
open DAT, "+< $db";       # Open for reading/writing.
open LS, "ls -l |";       # We read the output of 'ls -l'.
```

- Reading and writing from streams is easy.

```
my $line = <IN>;
print OUT "me: I just read $line\n";      # No comma.
```

- Lines read may contain line-ends.

```
chomp $line;
```

Regular Expressions

- You can use R.E. to check the format of strings.

```
print 'keyword' if $token =~ m/^(if|then|else)$/;
```

- But it is more interesting to extract data.

```
my @numbers = ($line =~ m/\d+/g);
while( my $line = <IN> ) {
        next if $line !~ m/([-.\w]+)@([-.\w]+)/;
        print "email: $1\@$2\n";
}
my ($dir, $file) = $path =~ m{(.+/)?([^/]+)};
```

Substitutions

- You can also use R.E. for search-and-replace.

```
my $msg = "She's wearing green.";
$msg =~ s/green/gold/;
print $msg;                 # prints She's wearing gold.
my $code = "if test then hiccup else wink";
$msg =~ s/(if|then|else)/\U\1/g;
print $code;    # prints IF test THEN hiccup ELSE wink
```

- There's also a similar utility for characters.

```
my $secret = "Don't tell anyone!";
$secret =~ tr/a-z/k-za-j/;    # Simple encryption.
print $secret;                # prints Dyx'd dovv kxiyxo!
```

Subroutines

- Arguments are not formal, Perl puts them in @_.

```
sub hypotenuse {
        my ($a, $b) = @_;
        return sqrt ($a**2 + $b**2);
}
sub sum {
        my $acc;
        $acc += $_ foreach @_;
        return $acc;
}
print sum(1, 2, 3, 4);       # prints 10
```

Reading FASTA files

```
open IN, "< $file" or die "Can't open $file: $!";
my $line = <IN>;   # Read the descriptor (ignored).
my $seq;
while( $line = <IN> ) {
        last if $line =~ /^>/;   # Stop if descriptor.
        $line = lc $line;
        $line =~ s/[^a-z]//sg;
        $seq = $seq . $line;
}
return $seq;
```

Translate DNA into RNA

```
sub dna_to_rna {
        my ($dna) = @_;
        $dna = lc $dna;
        $dna =~ s/[^acgt]//sg;
        my $rna = (reverse $dna) =~ tr/acgt/ugca/;
        return $rna;
}
```

Translate RNA into Protein

```
my %codonMap;    # ('gcu' => 'Ala', 'cgu' => 'Arg', ...)

sub rna_to_protein {
        my ($rna) = @_;
        my $protein;
        while( $rna =~ /(...)/g ) {
            $protein .= $codonMap{$1};
        }
        return $protein;
}
```

Initializing %codonMap

```
my %codonMap;

while( my $line = <DATA> ) {
        chomp $line;
        my @codons = split $line;
        my $residue = shift @codons;
        foreach my $c (@codons) {
            $codonMap{lc $c} = $residue;
        }
}
```

Other Points of Interest

- Modules
  - Build libraries of related subroutines which can be included with the *use* statement.
  - For an introduction see *man perlmod*.
- Object Oriented Programming
  - Perl supports OOP after a fashion. Robust, simple, and a bit off-putting at first, but you'll learn to like it.
  - For an introduction see *man perltoot*; if you're unfamiliar with OOP, start with *man perlboot*.
- Perl for Bioinformatics – BioPerl
  - http://www.bioperl.org/