Introduction to Perl

Andy Brown

Aims

- Be able to run a perl program
- Understand Variables and Data Structures
- Be able to write a program using
  - String manipulation
  - Number functions
  - Loops
  - Decision Making
  - Regular Expressions

Aims

- Be able to read and write a file
- Be able to write a perl one-liner
- Understand Functions
- Niceties if time:
  - Understand Modules
  - Understand Objects
  - Understand Testing

Please Ask!

Only the 2nd time!

Course Materials

- Modern Perl by Chromatic
- Slides
- Code
- These are for you to take away

How the course will run

- Sections
- Workshops
- Will use DNA sequence and Amino Acids where appropriate

Sections

- 00-Introduction
- 01-What is Perl?
- 02-Running a Program
- 03-Functions 1
- 04-Variables and Data Structures
- 05-IO
- 06-Regular Expressions

Sections

- 07-Decision Making
- 08-Let's Write a Script
- 09-File Operations
- 10-One Liners
- 11-References and Big Data Structures
- 12-Functions 2
- 13-Modules

Sections

- 14-Objects
- 15-Combining and when to use scripts/modules/objects
- 16-Testing
- 17-The End

Workshops

- Not many upfront
- Getting it right isn't as important as trying to do it
- We will go through all the workshop answers
- They will start to build on each other
- If you need help - Ask!

The Day

- 9.30am → 5pm sharp please
- 45 minutes for lunch?
- We will have tea breaks
- Please don't go back to your office at lunch

Rules

- Please don't spend your time checking emails or websites
- Please don't try to do work.

Who is this idiot?

- Andy Brown
- Been perl programming since 2002
- Full time since 2007
- I know some other languages
- Not formally qualified programmer!

Who are you wonderful people?

- Name
  - Please also fill in your name cards
- What do you do?
- What do you want out of this course?
- 3 words which describe you

What is Perl?

History

- Created in 1987 by Larry Wall, since Awk wouldn't do what he wanted
- Rapid development until Perl4 in 1991 and first release of The Camel
- Larry started work on Perl5 in 1993, with first release in 1994
- Perl5 has been active version ever since

History

- In 1995 the CPAN was created
- Jan 2011 – over 19,000 modules by over 8,000 authors
- Perl6 has been in development since 2000, but has stalled to a degree
- Perl5 now has yearly update, 5.12 released in 2010, 5.14 was released this year

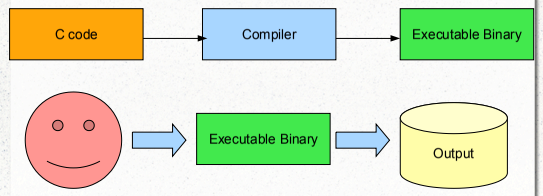
What does it mean?

- Backronym
  - Practical Extraction and Report Language
- Others have been suggested, including Larry's own
  - Pathologically Eclectic Rubbish Lister

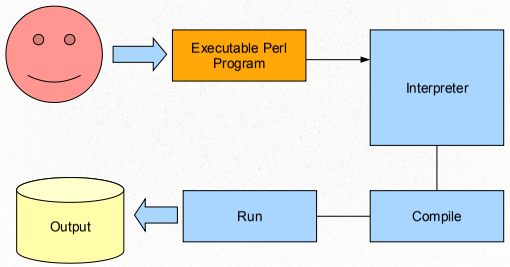
Language

- Perl is an Interpreted language
  - It needs to run through an interpreter to be compiled before execution

C program



Perl Program



Language

- Perl is a Dynamic language
  - Easy to change
  - Run code during compilation
  - Embed other languages within it
  - Can be embedded into other language

Language

- Perl is a Procedural language
  - Series of statements top to bottom - fast
- Perl is Dynamically typed
  - Variables checked at runtime, not compile time

Language

- Perl supports multiple programming paradigms
  - Object Oriented and Functional styles

Slogans

- There is more than one way to do it
  - TIMTOWTDI (TimToady)
- Easy things should be easy, and hard things should be possible

Applications

- Sysadmin support, GUI's, Web Apps, BioInformatics
- Plethora of reusable code on the CPAN to help with all of these
  - CGI for websites (as well as frameworks)
  - BioPerl (a multitude of Tools)

Licensing

- Open Source
- Free
- Encouraged to release useful code as Open Source (but not required)

Running a Perl Program

Aims

- Understand how to write a perl script
- How you run it

Shebang

- All scripts need to start with a hashbang (shebang) line (see bin/shebang.pl)

  #! /Users/ajb/dev/bin/perl (-wT)

- Lets take a look at this

#!

- Usually # signifies a comment in almost all languages
- Exception, first line found with a ! after it
- Tells the computer that this is the intepreter to use

/Users/ajb/dev/bin/perl

- The full file path of the interpreter executable
- This executable is responsible for compiling and running the rest of the script

(-wT)

- Optional arguments can be provided as though this were a command line
- w and T are the most common
- We will mostly be using w (warnings)
- T stands for Taint mode, and is used mainly on web applications

Lets run it!

- If we run

  ajb>bin/shebang.pl
- The computer executes

  /Users/ajb/dev/bin/perl -w bin/shebang.pl
- Try this

That didn't do anything!

- We now need to add some statements to execute
- The first one anyone ever writes for any language is hello_world

bin/hello_world.pl

- We add in a statement

  print 'Hello World!';
- Run it

  ajb>bin/hello_world.pl

  Hello World!ajb>

Lets look at the statement.

print

- print is a 'function' or 'keyword'. It tells perl to print the next argument

  'Hello World!'

- A string argument, that we want printed

  ;

- commands need a ; to signify that the command has ended – leaving it out will break your code

comments

- Look in the file

  bin/hello_world.pl

- *You'll see*

  # this is a comment to explain print (do this function on) 'Hello World!' (this string) ; (end of statement)

- Anything after a # until the end of the line is a comment, and is ignored

Failing scripts

- Perl tries to be helpful when scripts fail

  ajb>bin/fail_world.pl

  Illegal division by zero at bin/fail_world.pl line 4

- This is helpful. We know what the error is, and where it is in the script.
- Sometimes some investigation further is needed, but it helps.

External Code

- You can pull in extra code (modules) to provide extra functions

  use Modern::Perl;

  say 'Welcome, to a whole new world.';

- The *use* keyword tells the interpreter to go and locate this code, and compile and run it, and gives us *say*

External Code

ajb>bin/brave_new_world.pl

Welcome, to a whole new world.

ajb>

- Try removing the use line, save and retry
- BIG ERROR message – say function doesn't exist

Summary

- You must have the shebang line:

  #! /Users/ajb/dev/bin/perl (-wT)

- Statements follow

  <function> <parameters>;

- Use external code

  use Modern::Perl;

- Failures fail, and try to be helpful

Any questions so far?

Functions 1

Aims

- Understand what a function is
- What it does
- What it's benefit is

What is a Function?

- Functions are the verbs in statements
- These are sometimes referred to as keywords as they are part of core Perl
- They do something with some given inputs, and return out some outputs

What Benefits?

- They do a job for us
- Save us writing code
- Enable us to reuse code
  - Smaller code base
  - Less cutting and pasting

Is it a function, method or subroutine?

- All 3.
- Purists might tell you that there are differences, but it it mostly contextual.
- We will look at calling them by a term appropriate to context, but in code terms, they are all the same.
- So don't worry about it!

Function – Syntax

- We used a function already, print
  - *print 'Hello World!';*
- The simplest description of a function syntax would be

  return_value = function (list_of_parameters);
- Whilst the return value has been left out, it is the same.

Function – Syntax

- We also get functions which are called block functions, and look like this
  - ```
    while ( $x < 10 ) {
      print $x;
      $x++;
    }
    ```
- The syntax can be read in a similar way.

Functions

- Don't worry if you feel a little lost at the moment, it will come clear as we look at different functions.
- Common functions that we will come across include
  - open, close, if, next, last, foreach
  - while, until, unless, else ...
    ```

Variables and Data Structures

Aims

- Understand the different data types
- Manipulate numbers and strings with operators
- Tell the difference between scalar, array and hash

Data

- Functions generally need something to act upon - Data
- This can either be
  - Use once data – numbers and strings
  - Keep it around data - variables
- We will use some different functions than print/say. Please ask if I don't mention what the are trying to do, but we'll look more at some of them later

Numbers

- bin/01-integer.pl

  say 1;
- Integers don't need quoting

  say 0;
- 0 is just a number (well, actually its a concept...)

Maths

say 1+1; # addition

*say 2*7; # multiplication*

say 9-3; # subtraction

say 9/3; # division

- Your standard four maths operations

Maths

*say 3**2; # square*

*say 4**4; # (4 to the power of 4)*

- ** gives you to the power of

A string operation

say 1 . 1;

- We'll look at it when we do strings, but we have just concatenated these together

Boolean numbers

say 1 if 0; # this is boolean untrue

say 4 if 4; # any number other than 0 is true

- 0 is untrue, all other numbers are true (including negatives)

Floating Point

say 1.00; # truncates past the decimal point

say '1.00'; # acts as a string

say 1.01;

- Perl truncates trailing (and leading) 0's unless specified as a string

Producing Sequences

```
say 0..9;
```

- The double dot operator doesn't concatenate twice, but instead allows a sequence of consecutive numbers to be produced quickly

Numerical Testing

```
say 4 if ( 1 == 1 ); # lhv and rhv are equal
say 5 if ( 1 != 2 ); # lhv and rhv are not equal
say 6 if ( 2 > 1 ); # lhv greater than rhv
say 7 if ( 1 < 2 ); # rhv less than lhv
```

- = is an assignment operator in perl (and most languages) so we use == to check equivalence of numbers
- *also >= and =<*

Comparisons

```
say 10 <=> 11; # -1
say 10 <=> 10; # 0
say 11 <=> 10; # 1
```

- the comparison or 'spaceship' operator
- 3 possible results are given

  -1 if the right hand side is greater than left, 0 if the same,

  1 if the left hand side is greater.
- Useful if you want to sort a set of numbers.

Strings

- bin/02-string.pl

  ```
  say 'Hello World';
  ```
- Strings need to be quoted,

  ```
  say “Hello World”;
  ```
- Either single or double quotes

Length

```
say length 'Hello World';
```

- The length function gives us the length of the string in characters
- *You should note here, Perl is working right to left. 'Hello World' is passed to length, the output of this (11) is passed to say.*

Manipulating Strings

```
say lc 'TACtgATTACACATTACT'; # lower case the string
say uc 'tactgattacACATTact'; # upper case the string
say lcfirst 'TACTGATTACACATTACT'; # lower case the first letter
say ucfirst 'tactgattacacattact'; # upper case the first letter
```

Manipulating Strings

- User inputs name ANdrEw

```
say ucfirst lc 'ANdrEw';  # result Andrew
```

- So you can do some sanity checking!

Concatenate Strings

```
say 'Hello' . 'World'; # concatenate
```

- We saw concatenating two numbers previously. Now we are using strings.

Different Whitespace Characters

```
say "Hello\tWorld,\nHow are you today?"; # contains a newline and and tab character
```

- For different whitespace characters (tab/newline..) we can use standard character representations, but you must use double quotes.

Boolean Strings

```
say ''; # empty string, this is boolean false
say 'true string' if 'true'; # any string is true apart from the empty string
say 'true empty string' if ''; # this won't be shown, as empty string is false
```

- An empty string is false, everything else is true, incl space. (Uses ASCII value to denote truthfulness)

Producing Sequences

```
say 'a'..'z'; # quick alphabet
```

- Exactly the same as with numbers, you can use the double dot to generate a list.
- However, be careful if you meant to concatenate 'Hello'.'World'

```
say 'Hello'..'World;
```

Maths(?)

```
say 'a' * 2;
say 'a' + 'b';
```

- Not numeric warnings – ends up 0

```
say 'a' x 2;
```

- What you probably meant to do! Repeat a 2 times
- Note: In Perl number operators act on numbers, and Perl makes all non-number characters 0.

String Testing

```
say 'a equals a' if ( 'a' eq 'a' );
say 'a equals b' if ( 'a' eq 'b' );
say 'a does not equal b' if ( 'a' ne 'b' );
say 'a less than b' if ( 'a' lt 'b' );
say 'b greater than a' if ( 'b' gt 'a' );
```

- Because number operators only work on numbers, we need a string equivalent

String Comparisons

```
say 'a' cmp 'b'; # -1
say 'a' cmp 'a'; # 0
say 'b' cmp 'a'; # 1
```

- The same is true with comparison operators

```
say 'a' <=> 'b';
```

- not numeric warning, ends up 0 - equivalent

Number/String Comparison Table

	Number	String
Equal	==	*eq*
Not Equal	!=	*ne*
Less Than	<	*lt*
Gr Than	>	*gt*
Compare	<=>	*cmp*

Substrings

```
say substr 'TACTGATTACACATTACT', 0, 4;
say substr 'TACTGATTACACATTACT', 5, 12;
say substr 'TACTGATTACACATTACT', 5, 15;
say substr 'TACTGATTACACATTACT', -5, 3;
```

- As you'll see later, we use 0-based counting
- A negative offset counts from the end
- If you leave off the length, then you get all remaining characters

String Location

```
say index 'TACTGATTACACATT', 'GATTACA';
say index 'TACTGATCACATTACT', 'TACT', 7;
```

- We can get the location within a string of a character set
- Adding an offset skips over that many bases before searching

A math warning!

```
say 20/'b';
```

- This will break your code!
- Strings equate to 0, so this produces a divide by zero warning!

Other Things With Strings

- There are loads of other things to do with strings, such as regular expressions.
- We will come to these in time, but lets move onto holding strings and numbers around for longer - Variables.

Variables

- There are 3 basic types of Variable
  - Scalar – holds 1 piece of data
  - Array – holds a 'list' of pieces of data
  - Hash – holds a 'list' of named pieces of data
- Variables enable you to pass information around and operate on the data contained
- **Everything** you do in a program is likely to need to use a variable of some type

Scalar

- The simplest variable is a scalar
- A named 'box' containing one piece of data
- That data can be anything (more on that later)

bin/03-scalar.pl

$scalar

- Scalars are always denoted by a $ sign
  - Think of a special S as part of the name
- The first character of the name must be a letter
- The rest of the name can be
  - Letters
  - Numbers
  - Underscore

Naming

- Choose a name which represents what the variable should contain

  $sequence

  $amino_acid

  $height
- Not just some arbitrary names

  $one

  $foo

Assignment

- Use the = operator

```
my $integer_scalar = 10;
my $string_scalar = 'I am a string';
say $integer_scalar;
say $string_scalar;
```

Typing

- Perl uses dynamic typing, so this is OK

```
my $integer_scalar = 'I am a string';
```

- But DO NOT do it. Anyone looking at the code will be confused.

What can I do with it?

- Use a scalar exactly as though the number or string were there

```
say $integer_scalar + $integer_scalar;
say $string_scalar . $integer_scalar;
say $string_scalar * $integer_scalar;
```

What can I do with it?

```
say $string_scalar * $integer_scalar;
```

- This gave a warning. You probably wanted

```
say $string_scalar x $integer_scalar;
```

- x is a 'repeat' operator

Mix fixed and variable

- We can mix fixed data and variables

```
say 'I can use a fixed string and concatenate to ' .
$string_scalar;

say 30 / $integer_scalar;
```

String interpolation

```
say 'Hello, ' . $string_scalar;
```

- We don't have to concatenate these together

```
say 'Hello, $string_scalar'; # wrong
```

- Use double quotes

```
say "Hello, $string_scalar"; # right
```

- Double quotes interpolate variables, as wel as doing the right thing with \t,\n...

undef

- There is a 'magic' value **undef**
- All variables are assigned this if nothing else was assigned
- It is boolean **false**

```
my $undef_scalar;
say $undef_scalar
```

undef

- You can explicity assign undef

```
$string_scalar = undef;
say $string_scalar if $string_scalar;
```

- Only do this if you need to clear something out (say to recapture some memory after dumping the entire Lord of the Rings, with Appendices and The Hobbit to a scalar)

Arrays

- Arrays are a named group of scalars
- They are like a list
- Read top to bottom, or jumped to by an index
- Each position in the array is a scalar box, which is acted upon in exactly the same way as a scalar

bin/04-array.pl

Creating an Array

- Arrays are named with exactly the same rules as a scalar, except
  - They begin with an @ symbol
  - Usual to be a plural word
- They can be generated in a number of ways, but we still use the = assignment operator.

Creating an Array

```
my @string_array = qw{hello i am an array of strings};
```

- qw{} quotes each word, and we assign the quoted list to an array

```
my @integer_array = 1..10;
```

- As we saw before, 1..10 generates a sequence of integers, which we can assign to

Creating an Array

```
my @comma_made_array = ( 'i', 'see', 'a', 'little',
'silhoutte', 'of', 'a', 'man,' );
```

- Possibly the most common. Note the (), this tells perl to treat as an array of items

```
my @x_generated_array = ( 'scaramouche,' ) x 2;
```

- We can use the repeat operator to repeat an item multiple times, and then assign to the array

Array Operations

```
say 'The string_array, without any join: ' .
@string_array;
```

- Output directly concatenates all elements together.

```
say 'The integer_array, with a comma join: ' . join ',',
@integer_array;
```

- We can join them with a separator

Array Operations

```
say 'The fourth number in the integer array is ' . $integer_array[3];
```

- To obtain an individual element, change the @ to a $ (scalar), and then add the index with the [x] notation

```
say 'The last word in the string array is ' . $string_array[-1];
```

- Negative indices count from the end

Array Operations

```
my @slice = @string_array[1 .. 4];
say 'An array slice of the string array, indices 1->4: ' . join ' ', @slice;
```

- A subset of the array (slice) can be obtained by assigning to an array, with a sequence for the index

Array Operations

```
say 'The number of elements in string_array is : ' . scalar @string_array;
```

- length is string length, scalar is array length

```
say 'The length of x_generated_array is : ' . length @x_generated_array;
```

- The keyword scalar gives you the number of elements in the array. In some cases you can omit the word, but if you are always explicit, then you won't have a potential bugsite!

Interpolation

```
say 'The first word in the string array is $string_array[0]';
say "The first word in the string array is $string_array[0]";
```

- Just as we did with a scalar, we can insert into "" the scalar of an element in the array, and it will be interpolated into the string.

Doing things with Arrays

```
say join ' ', @comma_made_array, @x_generated_array, 'will you do the fandango?';
```

- Functions tend to expect arrays of arguments (we'll explain more later) but as such, we can use the comma to 'join' arrays together, so that the function treats as one array.

Doing things with Arrays

```
my @three_lines = ( ( join ' ', @comma_made_array ) , ( join ' ', @x_generated_array ), 'will you do the ballroom blitz?' );
say scalar @three_lines;
say join "\n", @three_lines;
```

- We can make new arrays from data processed

Iterating over Arrays

To iterate over an array, we can use the function foreach and act with each. Perl is basically taking each element in turn, assigning it's value to a scalar ($int/$str) and then you use this as the variable within the block of code (we'll look at code blocks later as well).

Iterating over Arrays

```
foreach my $int ( @integer_array ) {
  say $string_array[$int];
}
foreach my $str ( @string_array ) {
  say $integer_array[$str];
}
```

- Each element is assigned to a scalar, which we then use in the block of code

Empty Arrays

```
my @assigned_into_array;
```

- No need to explicitly do '= ()'

- Empty arrays are boolean false, as there are 0 elements.
- If there any elements the array is boolean true (incl. if they themselves are false)

Adding, Replacing and Removing Elements

```
$assigned_into_array[2] = 10;
```

- We can assign directly to an index. If there is already a value at that index, you will overwrite it.

```
say scalar @assigned_into_array;
```

- indices that are missing are autovivified, with *undef* as the value

```
say @assigned_into_array;
```

Adding, Replacing and Removing Elements

```
say scalar @comma_made_array;
push @comma_made_array, @x_generated_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- push adds elements to the end of an array. That can be a single element, or another array.

Adding, Replacing and Removing Elements

```
my $last_element = pop @comma_made_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- pop removes the last element of the array. Note it completely removes it, including the 'box' that it was in, so that the number of elements is also changed.

Adding, Replacing and Removing Elements

```
my $first_element = shift @comma_made_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- shift removes the first element of the array. Again, completely, so it will alter the length of the array, and also the other index positions

Adding, Replacing and Removing Elements

```
unshift @comma_made_array, $last_element;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- unshift puts elements onto the start of the array. It can be a single element or and array, and they will remain in that order.
- The indices of other elements will change

Adding, Replacing and Removing Elements

```
splice @comma_made_array, 3, 0, @x_generated_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- splice allows you to muck around with the internals of the array. This example adds one array into another, starting at index 3
- There are more examples in bin/04-array.pl

Hashes

- Unfortunately, a problem occurs with arrays which make it difficult to store lots of data. You need to remember the order the data is in

```
my @personal_info = ( qw{Andy Brown 180 brown blue} );
```

- I could have dyed my hair blue
- Depending on the country, Andy could be my family name

Hashes

- Fortunately, we have a solution – associative arrays, or hashes.

```
my %personal_info = (
  forename => 'Andy',
  surname  => 'Brown',
  height   => '180',
  hair     => 'brown',
  eyes     => 'blue',
);
```

Hashes

- We can then access the data with the name of the data (key)

```
say $personal_info{forename};
```

- *bin/05-hash.pl*

Creating a Hash

- Hashes are named with exactly the same rules as scalars, except
  - They begin with an % symbol
  - Plurality tends to depend on the information
- They can be generated in a number of ways, but we still use the = assignment operator.

2 Methods of Assignment

- You can create in two ways, 1 clearer than the other
- Unclear:

```
my %string_key_and_value_hash = qw{key1 val1 key2 val2 key3 val3};
```

- It is constructed in a similar way to an array, so could be confusing

```
# my %string_key_and_value_hash = ('key1', 'val1', 'key2', 'val2', 'key3', 'val3');
```

2 Methods of Assignment

- Clearer:

```
my %string_key_any_value_hash = (
  key1 => 'val1',
  key2 => 'valB',
  key3 => '3',
  key4 => undef,
);
```

- => acts like a comma (the fat comma)
- Always use this form!

Outputting data

```
say %string_key_any_value_hash;
```

- The output of this is in a random order, which is due to the internal way perl stores the data

```
foreach my $key ( sort keys %string_key_any_value_hash ) {
  say $key . ' : ' . $string_key_any_value_hash{$key};
}
```

- keys %x gives an array of the keys only, which we sort and then use in foreach

Interpolation

```
say 'The first value in the string_key_and_value_hash is $string_key_and_value_hash{key1}';
say "The first value in the string_key_and_value_hash is $string_key_and_value_hash{key1}"; # again, double quotes interpolate the variable
```

- As before, we can interpolate the values from a hash using double quotes directly into a string

Keys can be Numerical

```
my %keys_are_numbers = (
  1 => 'one', 2 => 'two', 3 => 'three',
);
say $keys_are_numbers{2};
```

- There are no real benefits to this, as arrays are faster, except possibly
  - You don't have to start at 0
  - You won't autovivify the missing indices if you want to assign to a later number

Summary

- Strings and Numbers can be processed, by various functions and operations
- Scalars are the basic 'storage' unit in a perl program, and contain 1 string or number
- Arrays are lists of elements (scalars) that are ordered by index
- Hashes (associative arrays) contain scalars which are named and accessed with a key

Inputs and Outputs

Aims

- Understand the main ways of obtaining data into a program that are not file/db based
- Have an idea about the two main outputs
- Be able to write a script which can take in an a user input, do something to it, and output it to screen

Inputs - ARGV

- The ARGV array is the array of arguments that you pass in on the command line

  bin/01-argv.pl
- Run the above script with some arguments

  >bin/01-argv.pl I love perl

  I

  love

  perl

Inputs - ARGV

- Our arguments can be obtained via array

  @ARGV
- Always assign this array to an array you define in your code!

  my @arguments = @ARGV;

Inputs - ARGV

- We can then do as we like with the array

```
foreach my $argument ( @ARGV ) {
  say $argument;
}
```

- This is by far the simplest, most convenient way of getting any data in. People expect it, you don't need to worry about interactivity.

Inputs - STDIN

- The command line is very nice. However, it's no good if you want to interact with your users
- For this we use the STDIN filehandle. A filehandle is a way that the program interacts with a source of data whilst running.
- STDIN is a filehandle for input from the keyboard (usually)

Inputs - STDIN

- Run

  >bin/02-stdin.pl

  Enter a number: 12

  You entered 12. The square of this number is 144.

Inputs - STDIN

print 'Enter a number: ';

- We ask the user for something

  my $number = <STDIN>;
- <STDIN> is the filehandle, and tells perl to wait for something from the keyboard, followed by a newline (enter).
- We then must assign it to a variable, or it will be lost.

Inputs - STDIN

chomp $number;

- We take off the newline with the chomp method, since we want a number only

  *say "You entered $number. The square of this number is " . $number**2 . '.';*
- Feedback to the user their number, and do something with it.

Inputs - DATA

- Sometimes, it is worth having the input data (or some at least) in the file with the code – when unlikely to change much!
- The obvious thing would be to put this data in the code, and directly assign

  my @info = qw{hair eyes nose chin};
- But, it isn't very easy to find, or for a non-programmer to read.
- Solution: the DATA filehandle and tag

Inputs - DATA

- Run

  >bin/03-data.pl

  Is this the real life?

  Is this just fantasy?

  Caught in a Landslide

  No escape from Reality ...
- Song lyrics don't tend to change much, but they would clutter the code up

Inputs - DATA

```
while ( my $line = <DATA> ) {
  print $line;
}
```

- <DATA> is a special filehandle which says to read all the text after either of the following special markers: *__DATA__* or *__END__*
- The perl parser knows there is no more code after either of these markers

Inputs - DATA

__DATA__

Is this the real life?

Is this just fantasy?

Caught in a Landslide

No escape from Reality ...

- So, after the marker, we can write any text, and perl will look at it only once the <DATA> filehandle is used, as input

Outputs - STDOUT

- STDOUT is the generally the window you are working in.
- Run

  bin/04-stdout/pl

  I am saying to STDOUT

  I am printing to STDOUT

Outputs - STDOUT

- If we look at bin/04-stdout.pl

  say 'I am saying to STDOUT';

  print "I am printing to STDOUT\n";
- say and print both go by default to STDOUT without the need to specify a filehandle

Outputs - STDERR

- STDERR is the filehandle that errors go to.
- Run:

  >bin/05-stdout.pl

  I am warning to STDERR at bin/05-stderr.pl line 5.

  I am dying to STDERR at bin/05-stderr.pl line 6.
- By default, STDERR inherits from the shell its error location

Outputs - STDERR

- Lets look at bin/05-stdout.pl

  warn 'I am warning to STDERR';

  die 'I am dying to STDERR';
- The warn and die methods both output to STDERR
- The difference between STDOUT and STDERR happens if you decide to overide STDERR to output to a file (e.g. server log), or an environment setting changes it (e.g. LSF options)

Task

- Spend the next few minutes writing a script which
  - Has 2 inputs from different methods
  - Does something with those inputs
  - Outputs the result of what you did
- We will take a look at a couple.
- It doesn't matter what input methods of the 3 you use, or what you do to process and change them (look back at string/numbers in previous section)

Regular Expressions

Aims

- Understand what a regular expression is
- Use match, substitute and transliterate
- Know about using i,m,g flags
- Understand some metacharacters
- Capture
- Greedy and non-greedy match
- Count with transliterate

Regular Expressions

- Perl's speciality
- Heavily optimised to be as efficient as possible
- Reliable
- Fully featured (as yet, I haven't found anything it is missing)

Regular Expressions

- 3 types of regular expressions
- Match
  - Does the variable contain
- Substitute
  - If match, then replace with something
- Transliterate
  - Swap characters

Match

- Problem: I want to see it a sequence contains the codon for Methionine
- So far, we could look to see if there is an index returned from

  say index $seq, 'ATG'
- However, that isn't very obvious, and what if the sequence is lower case?

  bin/01-match.pl

Match

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
say $sequence;
if ( $sequence =~ m/ATG/i ) {
  say 'Sequence contains Methionine';
}
```

- This tests to see if it can find ATG at all in the sequence, and lets us know if true

Match - Syntax

- Lets break down the match

  if ( ) {

  }
- This is the block form of if, we'll look at that more in the next section. For now, accept that it is just allowing us to 'visualise' the match

Match - Syntax

$sequence =~ m/ATG/i

- This is the syntax for a match

  $sequence
- Our variable to act upon

  =~
- The 'true' match operator. The return value of the expression is a boolean, so this sets match as boolean true, fail as boolean false

  (No match to be true would use !~)

Match - Syntax

$sequence =~ m/ATG/i

m/ATG/i

- m tells perl we want to do a match regular expression operation
- /xxx/ delimits the thing we want to match
- i is a flag options

Match

- Obviously, we can just swap ATG for whatever we want to search for

  AAA

  Hello World

  01223834244

  \n (\t)
- There are also a set of metacharacters we can use, to match out of a set.

Match - Metacharacters

```
if ( $sequence =~ m/\d+/ ) {
  say 'Sequence contain a number';
}
```

- \d matches a digit
- + matches 1 or more of the preceeding character or metacharacter

```
if ( $sequence !~ m/\d+/ ) {
  say 'Sequence does not contain a number';
}
```

Match - Metacharacters

```
if ( $sequence =~ m/\w+/ ) {
  say 'Sequence contains word characters';
}
```

- \w matches a word character
  - Alpha
  - Underscore
  - Digits

Match - Metacharacters

```
my $lyrics = 'I see a little silhoutte of a man';
if ( $lyrics =~ m/\s/ ) {
  say 'Lyrics contain whitespace';
}
```

- \s matches a whitespace character
  - Space
  - Line break
  - Tab

Match - Metacharacters

- These are the most common metacharacters you will see along with

  \A – match the start of the string

  \z – match the end of the string

  . - match any character

Match - Capture

- As well as matching, you can capture part or all of what you are trying to match, by placing brackets around the bit to capture

```
my ( $silhoutte ) = $lyrics =~ m/\s(silhoutte)\s/i;
say $silhoutte;
```

- In this case, if we match the word silhoutte, surrounded by whitespace, then put that word into the variable

Match - Capture

- How about capturing the first word:

```
my ( $first_word ) = $lyrics =~ m/(\w+)\s+/;
say $first_word;
```

- Or the last:

```
my ( $last_word ) = $lyrics =~ m/\s(\w+)\s*\z/m;
say $last_word;
```

- * means zero or more

Match - Greedy

- * means zero or more, + means 1 or more
- However, they like to grab more than less

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
my ( $greedy_match ) = $sequence =~ /(\w+ACC)/;
say $greedy_match;
```

- How do we stop this?

Match – Non-Greedy

- Use the ? metacharacter

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
my ( $non_greedy_match ) = $sequence =~ /(\w+?
ACC)/;
say $non_greedy_match;
```

- ? tells the regex to stop matching after the first it can match

Match – User defined

- Our user wants to see if their favourite codon is present

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
my $user_wants_to_find_this_sequence = 'ttt';
```

- How?

Match – User defined

- We can put their variable in the match

```
if ( $sequence =~ /
$user_wants_to_find_this_sequence/i ) {
  say "User sequence
$user_wants_to_find_this_sequence found.";
}
```

- The regular expression is allowed to contain variables.
- Much more useful than forcing only to match developer designated expression

Match – i and m

- What do the i and m mean?
- They are flags which tell the regex to work in particular ways (note, these are here for completeness, not because you need to learn them all)
- i – this tells the regex to ignore case

Match – i and m

- m - makes perl match newlines/end of strings the same as other languages
- In most languages:
  - ^ matches the start of a line, but in perl the start of the string
  - $ matches the end of a line, but in perl the end of a string
- This makes it a little confusing, so m changes their meaning
- Example at end of bin/01-match.pl

Substitute

- Problem: We want to replace a codon with another in the sequence.

bin/02-substitute.pl

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
unless ( $sequence =~ m/cat/i ) {
  say 'we do not have a cat';
}
```

Substitute

- Lets replace cac with cat

```
$sequence =~ s/cac/cat/i;
if ( $sequence =~ m/cat/i ) {
  say 'we now have a cat';
}
```

- How many 'cat's?

```
say $sequence;
```

- 1, but there were 3 'cac's

Substitute - Syntax

- Lets look at the syntax, comparing with match

```
$sequence =~ s/cac/cat/i;
$sequence =~ m/cat/i
```

- =~ That's the same. We are informing perl that we wish to perform a regex operation on $sequence

Substitute - Syntax

```
$sequence =~ s/cac/cat/i;
$sequence =~ m/cat/i
```

- s instead of m
  - We tell perl that we want to do a substitution type of regex
- i – as before we want to do case insensitive matching

Substitute - Syntax

```
$sequence =~ s/cac/cat/i;
$sequence =~ m/cat/i
```

- /*match*/*replace*/ instead of /*match*/
  - match is identical, and you can use all the same matching tech
  - But, perl needs to know what to replace the match with, so we put that in a second 'column'

Substitute 'g'lobally

- How many 'cat's?

  say $sequence;
- 1, but there were 3 'cac's
- It only substituted the first one it found!
- How do we replace all of them?

  $sequence =~ s/cac/cat/gi;

  say $sequence;
- The g flag tells perl to keep matching and substituting until the end

Substitute – User defined

- You can have variables as the match, and the replacement

  my $favourite_codon = 'aaa'; # it's all the first letter of my name;

  my $least_favourite_codon = 'cat'; # I don't actually like cats

  $sequence =~ s/$least_favourite_codon/ $favourite_codon/gi;

  say $sequence;

Substitute – How many Substitutions made?

- The return value is the number of substitutions made, or undef if 0

  say $sequence =~ s/aaa/cac/i; # 1 (we didn't specify global)

  say $sequence =~ s/aaa/cac/gi; # 2 (the remaining ones)

  say $sequence =~ s/cat/aaa/gi; # undef, we had already cleared away all the cats
- This means you can use it as a boolean

Substitute – Don't be greedy!

- Because the match is exactly the same as with a match regex, + and * are still greedy

  my $clone = $sequence;

  *$sequence =~ s/g.*cac/aaa/i;*

  say $sequence;

  $clone =~ s/g.?cac/aaa/i;*

  say $clone;
- You would not have been able to do multiple substitutions, as the first match ate everything between the first g and last cac

Match and Substitute

- There are other metacharacters, and ways of constructing really powerful short regexes with other flags
- However, everything you have seen here will form the basis of any match you do, and you should be able to construct virtually any match you want
  - 90% of my regexes use nothing more than shown here

Transliterate

- The 3rd type of regex is transliterate. It stands a little aside from match and substitute, as it is more literal (i.e. less metacharacter and flag driven)
- Problem: I want to know the reverse strand of my DNA sequence
- You can't use substitute easily
  - s/T/A/gi
  - s/A/T/gi

Transliterate

bin/03-transliterate.pl

my $sequence = 'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG CGCCGTCACGCTCGCAGTC';

my $reverse_strand = $sequence;

$reverse_strand =~ tr/ACGT/TGCA/;

say $sequence;

say $reverse_strand;

- In-place substitution of the dna letters

Transliterate - Syntax

- Let's look at the syntax

  reverse_strand =~ tr/ACGT/TGCA/;
- It's quite similar to substitute, but there are important differences
- =~ We now know what this means
- tr – tells perl to perform a transliterate operation on *$reverse_strand*

Transliterate - Syntax

reverse_strand =~ tr/ACGT/TGCA/;

- /ACGT/TGCA/

  This is the important bit. Whereas before we had match 'patterns' and substitution strings, we don't here.
- As it reads *$reverse_strand*, each time it finds a letter in the first part, it replaces with the equivalent letter in the second
  - Find an A, replace with T
  - Find a G, replace with a C

Transliterate – Return Value

- As with substitute telling you the number of substitutions made, transliterate returns the number of transliterations made (or undef)

```
$reverse_strand = $sequence;
say $reverse_strand =~ tr/ACGT/TGCA/;
my $bees = 'BBBBBBBBBBBBB';
say $bees =~ tr/ACGT/TGCA/;
```

Transliterate – Coding a Message

- A bit of fun, lets use this to apply a simple cypher

```
my $original_message = 'The quick brown fox jumped over the lazy dog.';
say $original_message;
my $coded_message = $original_message;
$coded_message =~ tr/a-zA-Z ./N-ZA-Mn-za-m. /;
say $coded_message;
```

- In regexes, we can shorten sequences with a dash, they get expanded by the parser.

Transliterate – Counting Characters

- You can replace a character with itself, to get a count of that character in the string, without changing the string

```
say $sequence;
say "The number of A's in above = " . ( $sequence =~ tr/Aa/Aa/ );
say $sequence;
```

- It is unchanged, but we know how many A's are present

Transliterate – GC percentage

- With a bit of playing, you could work out the percentage GC content

```
say "percentage GC = " .
        ( ( $sequence =~ tr/CGcg/CGcg/ ) * 100 /
         (  $sequence =~ tr/ACGTacgt/ACGTacgt/ ) );
say $sequence;
```

- Again, unchanged, but we know what the GC percentage is

Quick play with regexes

- Lets just spend a few minutes playing with regexes.

  Produce a regex which will capture a name and a phone number into variables from the following:

  Andy Brown: 01234 567890

- Bonus points for
  - Using metacharacters
  - name parts and phone parts

Decision Making and Looping

Aims

- Understand if/elsif/else decision block
- Be able to use a foreach and while loop
- Understand and/or/not &&/||/!
- Using comparisons for decisions

Decision Making

- Programs are going to be pretty unhelpful long term if they don't at least try to do different things, dependent on data given.
- Contrary to what some people might think, computers are stupid, and will only do what you tell them. So, we need to tell them.
- Even perl programs

  bin/01-if_else.pl

Decision Making -if

- 'if' will be probably be the most called function you will ever write
- There are two ways of using it, the block form and the postfix form
- Let's take a look at them

Decision Making -if

- The block form

```
my $this = 'Hello';
my $that = 'Hello';
if ( $this ) {
  say $this;
}
```

- You can see that we are passing an array ( ) of arguments, the conditional, to *if*, which if true executes the block of code { }

Decision Making -if

- The postfix form

```
say $that if $that;
```

- Here we are passing an argument to *say*, and then also telling it to only *say* the argument if the conditional argument passed to *if* is true
- A word of advice, keep usage of postfix if to a minimum – **only** when very simple action and test

Decision Making - unless

- There is a negative version of if, unless. It has fallen out of favour, but can sometimes save you a lot of 'not's in your code

```
my $false;
unless ( $false ) {
  say 'I tested false';
}
say '0 is false' unless 0;
```

- Reason for not using, no elsunless block

Decision Making - else

- You are probably going to want to do something *else* if the *if* wasn't true.

```
my $seq = 'CCGGATCACTATGACCTGCTTTCG';
if ( $seq =~ m/ATG/ixms ) {
  say "$seq contains Methionine";
} else {
  say "$seq contains that damned cat again";
}
```

- This matches, so only the *if* block happens

Decision Making - else

```
my $clone = $seq;
$clone =~ s/atg/cat/gim;
if ( $clone =~ m/ATG/im ) {
  say "$clone contains Methionine";
} else {
  say "$clone contains that damned cat again";
}
```

- We have removed the ATG, so since it doesn't find that, we execute the else block

Decision Making - elsif

- So we have the possibility of doing 2 things, but can we do multiple options
  - Yes
- elsif allows us to put in extra optional truth statements

Decision Making - elsif

```
$clone =~ s/dog/aaa/gim;
if ( $clone =~ m/ATG/im ) {
  say "$clone contains Methionine";
} elsif ( $clone =~ m/dog/im ) {
  say "$clone from foreign planet, or this isn't DNA";
} elsif ( $clone =~ m/aaa/im ) {
  say "$clone has my favourite codon in it";
} else {
  say "$clone contains that damned cat again"; }
```

Decision Making - elsif

- You can have as many or as few *elsif* blocks as you like
- Each must have a condition and a block
- The tests stop as soon as a condition is true

```
$clone .= 'atg';
if ( $clone =~ m/ATG/im ) {
  say "$clone contains Methionine";
} elsif ( $clone =~ m/dog/im ) {…} … # never tested
```

Looping

- There are 4 loop keywords. *while*, *until*, *for* and *foreach*.
  - *while* and *until* are conditional based
  - *for* is iteration and conditional based (and less commonly used)
  - *foreach* is iteration based
- All can be rewritten to do the same as each other

bin/02-while_foreach.pl

Looping - while

```
my $true_comparison = 10;
my $count = 0;
while ( $count < $true_comparison ) {
  say 'while version : ' . $count;
  $count++; # increment count
}
```

- The syntax is similar to an *if* statement. The *while* keyword, followed by a conditional, and then a block of code

Looping - while

- The difference comes that *if* only executes it's block once.
- Every time the end of the block is reached, *while goes back to see if the conditional is still true,* and then re-executes until the conditional is false

```
while version : 0
...
while version : 9
```

Looping - until

- *until* is related to *while* in the way *unless* is to *if*

```
$count = 0;
until ( $count == $true_comparison ) {
  say 'until version : ' . $count;
  $count++;
}
```

- *until* keeps executing the block, whilst the conditional is false
- *until* is the least favoured of the two

Looping – while and until

- A warning – don't do the following, unless you have very good reason

```
while (1) {
…
}
until (0) {
…
}
```

- Both will never stop looping!

Looping – foreach

- When you have an array of elements that you want to process, you can use *foreach*

```
my @codons = qw{AAA AAC AAG AAT ACA ACC
ACG ACT AGA AGC AGG AGT ATA ATC ATG ATT};
foreach my $codon ( @codons ) {
  say 'foreach version : ' . $codon;
}
```

- Foreach element (codon) in the array, assign to the $codon variable, execute the block with that variable

Looping – for

- *foreach* doesn't provide you with the index of the element in the array
- To do this, we use *for*

```
for (my $i = 0; $i < @codons; $i++ ) {
  say 'for version : ' . $codons[$i];
}
```

- This a combination of a conditional and an iteration

Looping – for

```
for (my $i = 0; $i < @codons; $i++ ) {}
```

- my $i = 0;
  - Start index variable with first index
- $i < @codons;
  - Conditional, keep going whilst $i < array length
- $i++
  - Increment index variable after each execution

Looping – for and foreach

- *for* and *foreach* are actually interchangeable due to historical wishes to be similar to other languages
- However, it is more common to see *for* replace *foreach* than the other way around

Looping – with instead of foreach

- You can write the code to use any one of the four to do all of the jobs e.g.

```
while ( my $codon = shift @codons ) {
  say 'while method of looping over array: ' . $codon;
}
```

- However, be warned, this would empty the @codons array, as it's 'shift'ing the codon
- The conditional fails when there is nothing to assign

Boolean Logic Operators

- Often, you are going to want to have more than one condition to be true (or false) in order for an action to occur
- We have access to the usual Boolean Logic operators to help here
  - *&& / and* = and
  - *|| / or* = or
  - *! / not* = not

bin/03-and_or_not.pl

Boolean - &&/and

- If you want an action when two or more conditions are true

```
my $true = 1;
my $also_true = 2;
my $false = 0;
my $also_false = 0;
if ( $true && $also_true ) {
  say 'true && also_true is true';
}
```

Boolean - &&/and

```
unless ( $true && $false ) {
  say 'true && false is not true'
} # note: using unless
unless ( $false and $also_false ) {
  say 'false and also_false not true';
}
```

- *&&* (or *and*) is looking for both parts to be true, and then makes the overall conditional true if all parts are

Boolean - ||/or

- If you want an action when at least one of the conditions is true

```
if ( $true || $also_true ) {
  say 'true || also_true is true';
}
if ( $true || $false ) {
  say 'true || false is true';
}
```

Boolean - ||/or

```
if ( $false or $true ) {
  say 'false or true is true';
}
unless ( $false or $also_false ) {
  say 'false or also_false is false';
}
```

- || (or *or*) is looking for either part to be true, and then makes the overall conditional true
- Shorts out at the first one it finds to be true

Boolean - !/not

- You may want to invert the truthfulness of something.

```
if ( ! $false ) {
  say 'false has been made true with !';
}
if ( not $also_false ) {
  say 'also_false has been made true with not';
}
```

- *!/not* always turns values to undef or 1

Boolean – Complex Conditionals

- We can use Boolean Logic to create complex conditionals
- Use () to group parts of conditionals

```
if ( $false
      ||
    ( $true && $also_true )
  ) {
  say 'I got to true in the end';
}
```

Boolean – Complex Conditionals

- does sequence contain a methionine and stop (may be a gene)

```
my $sequence = 'CCGGATCACTATGACCTG';
if ( $sequence =~ m/atg/im &&
    ( $sequence =~ m/TAA/im || $sequence =~ m/TAG/im || $sequence =~ m/TGA/im )
  ) {
   say 'contains a methionine codon and a stop codon';
}
```

Boolean – Complex Conditionals

- Or contains Methionine but not Glutamine

```
my $sequence = 'CCGGATCACTATGACCTG';
if ( $sequence =~ m/atg/im
     &&
    ! ( $sequence =~ m/CAA/im || $sequence =~ m/CAG/im )
  ) {
   say 'contains a methionine codon and not a glutamine codon';
}
```

Boolean – Complex Conditionals

- Note: either use &&/||/! or and/or/not, don't mix and match unless you have reason – they have different precedences and you will more than likely introduce a but
- So, as a rule of thumb, pick one, and stick with it.

Boolean – Assignment

- The || operator is really useful in variable assignment

```
my $truth = $false || $true;
say 'truth: ' . $truth;
```

- line up many, cut out as soon as a true value found

```
$truth = $false || $also_true || 'I am a truthful string';
```

- With the above you can give preferential values to a variable

Boolean – Assignment

- ||= (or assign) is an exceptionally common piece of code

```
$truth ||= $true;
say 'truth: ' . $truth;
$truth = 0; # remember, 0 is false
$truth ||= {};
say 'truth: ' . $truth;
```

- We'll see more of this sort of thing a lot when we explore functions and objects

Comparisons

- We have seen straight forward depending on truth, and regexes as our conditionals
- Don't forget, we can use any comparison tests as a conditional
  - Equal
  - Not Equal
  - Less than
  - Greater than...

bin/04-comparisons.pl

Comparisons - Equal

```
if ( 10 == 10 ) {
  say 10 == 10;
}

if ( 'cat' eq 'cat' ) {
  say 'cat' eq 'cat';
}
```

Comparisons – Not Equal

```
if ( 10 != 20 ) {
  say 10 != 20;
}

if ( 'cat' ne 'dog' ) {
  say 'cat' ne 'dog';
}
```

- you can continue for >< lt gt => <=

Decision Making and Looping

- Decisions and looping are two fundamental parts of programming.
- You need to instruct the program how to deal with the data given, and if there are choices to make, what rules to use.
- Hopefully, we have covered the syntax here so that you can at least write (or understand) some of these, and have some decisions made.
- The next workshop should help

Workshop
Let's Write a Script

Let's Write A Script

- Folder 08-lets_write_script
- All the instructions are in file

  bin/01-sequence_play.pl
- Along with some sequences already assigned to variables
- Work through each of the questions in the file (doing your work in it).
- There are challenges relating to all the 4 sections we have covered.
- *there is more than one way to do it*

Answers

- 1) lower case each string

```
say lc $sequence_1;
say lc $sequence_2;
say lc $sequence_3;
```

Answers

- 2) do any of the sequences contain methionine (atg)?

```
if ( $sequence_1 =~ m/atg/ixms ) {
  say 'sequence 1 contains methionine';
}
```

Answers

- 3) do any of the sequences contain a stop codon (taa,tag,tga)?

```
if ( $sequence_1 =~ m/taa/im || $sequence_1 =~ m/tag/im || $sequence_1 =~ m/tga/im ) {
  say 'sequence 1 contains stop';
}
```

Answers

- 4) what is the %age GC content of each string?

```
foreach my $seq ( $sequence_1, $sequence_2, $sequence_3 ) {
  say ( ( ( $seq =~ tr/GCgc/GCgc/ ) * 100 ) / length $seq );
}
```

Answers

- 5) Let the user enter a codon to exchange methionine for in sequence_1, and switch it

  Bonus points for challenging a non-dna base letter

Answers

```
print 'Enter a codon you would like to swap methionine (atg) for: ';
my $user_codon = <STDIN>;
chomp $user_codon;
while ( $user_codon =~ m/[bd-fi-su-z]/im ) {
  print 'The codon you entered contains non_dna bases. Please try again: ';
  $user_codon = <STDIN>;
  chomp $user_codon;
}
$sequence_1 =~ s/atg/$user_codon/im;
say $sequence_1;
```

Answers

- 6) BONUS: assuming sequence is always read 5' to 3', can you print out the reverse complement in the order it is read of each sequence

```
foreach my $seq ( $sequence_1, $sequence_2, $sequence_3 ) {
  $seq =~ tr/ACGTacgt/TGCAtgca/;
  $seq = reverse $seq;
  say $seq;
}
```

File Operations

Aims

- Understand how to read a file into a program
- Understand how to write out data to a file
- Why File ::Slurp is good
- What a filehandle is, and how to use one
- How to append as well as write a new file

File operations

- Your end user is not going to want to provide everything on the command line, or permanently interact with a running script
- Also, they are not going to want to 'screen scrape' the end data
- Most data will come from/stored into a file or a database
- Database interaction beyond the scope of this course

File ::Slurp

- There is a great module of code you can use to help with read and write file operations

  use File ::Slurp;
- We get the methods *read_file* and *write_file*
- We'll look at both, and the filehandle versions to compare

Reading a file – File ::Slurp

- Let's start with reading a file

  bin/01-reading_a_file.pl
- Slurp in a whole book

  my $slobbit = read_file('data/the_slobbit');
- Now we have some data in the program, we can view it

  say $slobbit;

Reading a file – File ::Slurp

- we can do something to it
- Nobody wants me to be the hero of the story - *$slobbit* is just a string

  $slobbit =~ s/Andy/Bilbo/gm;

  say 'Hero changed';

  say $slobbit;
- but we haven't changed the file - take a look at *data/the_slobbit*

Reading a file – File ::Slurp

- our data may be in some table like format, so we probably don't want it all in one scalar
- slurp in a qseq file, with rows going into an array

  my @reads = read_file('data/1234_1_qseq.txt');

  say @reads;

Reading a file – File ::Slurp

- We want just the DNA sequences

```
foreach my $read ( @reads ) {
  my @data = split /\s+/m, $read;
  say $data[8];
}
```

- In both of these cases, we are going to need to watch the memory, as all the file is being read in at once.

Reading a file - Filehandle

- In many cases, always take advantage of File::Slurp
- However, if you want to do it completely by yourself, you need to go through some steps and open a *filehandle*
- A *filehandle* is a *scalar reference* to the file, which allows you to read/write to the file

Reading a file - Filehandle

- open a file handle to the file you want to read

```
open my $fh, '<', 'data/the_slobbit' or die 'Could not open data/the_slobbit for reading';
```

- *$fh* is a file handle
- *'<'* means for reading only
- *'data/the_slobbit'* is the filename
- always give an option to do something if the file can't be opened, in this case die

Reading a file - Filehandle

```
while ( <$fh> ) {
  s/Gran/Uncle/xms;
  print;
}
```

- The file handle can act just like an array, so we can (for example) loop on it, processing a line at a time

Reading a file - Filehandle

```
close $fh or die 'Could not close the filehandle for file data/the_slobbit';
```

- You must close the filehandle, with an option to do something if it can't close (this is unlikely to happen)

Reading a file - Filehandle

- The advantage of this is that you will only read out of the file the next line to process, so in the case of a million+ line fastq file, you won't fill your memory up.
- The disadvantage is that your files may become altered/vanish/locked during this time, causing problems either for your program, or other users.

Writing a file

- we looked at the data source file, but you are going to want your results to go somewhere else, and are likely to want to store them in a file.

  bin/02-writing_a_file.pl
- In this script is an array of sequences that we will want to write out to a file (they would have been generated in some way)

Writing a file – File::Slurp

```
write_file( 'data/short_reads.seq', @sequences );
```

- check the file *data/short_reads.seq*. You'll see all the reads are now in there, ready to be passed on, archived, processed further...

Writing a file – filehandle

- As with reading, there is the full way of doing it. This is very useful if you need to write periodically (e.g. a log file) or you need to write specific data, or just modify data in the file
- first, direct equivalent to write_file

Writing a file – filehandle

- open a file handle to the file you want to write to

```
open my $fh, '>', 'data/my_book' or die 'could not open data/my_book';
```

- *$fh* as with reading, we need a file handle
- *'>'* this means for writing, overwriting any existing file with this name
- *'data/my_book'* the file we want to write into
- *or die* again always error handle an open

Writing a file – filehandle

```
foreach my $seq ( @sequences ) {
  print {$fh} 'Andy ' . $seq or die 'Unable to print to filehande: ' . $fh;
}
```

- In this loop, as we process the elements, we print to the filehandle our data.
- Advantage of this, if it takes a lot of processing to create each sequence, then we can print as we generate, rather than collect them all in one go

Writing a file – filehandle

print {$fh} 'Andy ' ;

- we put the filehandle in *{ }* so that the interpreter knows that it is 'where to print to', rather than needing to check at runtime if it is a variable to print, or something to print to

close $fh or die 'could not close filehandle: ' . $fh;

- Exactly as with reading, we close the file handle

Appending to a file – filehandle

- If you want to append to a file (example, you have a script which runs every hour on a cron, then you don't want to kill data generated in previous hours)

open $fh, '>>', 'data/my_book' or die 'could not open data/my_book';

- *'>>'* this means for appending, creating the file if it did not already exist

Appending to a file – filehandle

foreach my $seq (@sequences) {

print {$fh} 'James ' . $seq or die 'Unable to print to filehande: ' . $fh;

}

- We'll add James this time, so that we can see it has been added

close $fh or die 'could not close filehandle: ' . $fh;

- We close again

File Operations - Summary

- We have seen how to read a file into the program with write_file from File ::Slurp and using a filehandle
- We have seen how to write to a file from the program with read_file from File ::Slurp and using a filehandle
- We have seen how to append to a file
- We have touched upon when a filehandle might be better – large amounts of data

File Operations - Workshop

- 09-file_operations/workshop
- bin/01-reading_and_writing_workshop.pl
- using the file given file write a script that will obtain the sequence and the quality scores, generate a unique name for each read from other information, and then output out in the format
- # read_name
- # sequence
- # quality
- into another file in the output directory

One-Liners

Aims

- Know what a perl one-liner is
- Know how to use one
- Know when they are useful

What Are One-Liners?

- Perl one-liners should be mentioned in an introductory course.
- Now, depending on how you look at it, these are
  - the most useful things possible,
  - or the worst idea ever imagined.
- They are exactly what they say on the tin. A perl script, in one line, on the command line.

Our first One-Liner

- Just try typing the following on the command line:

  >perl -le 'print q{Hello World}'

  Hello World

  >
- Let's take a look at it

Our first One-Liner

perl

- The first argument of any command line is the executable you want to run. In this case, we'll run the perl interpreter.
- This is equivalent to the shebang line telling the os to run perl
- This will use the first perl on our path

Our first One-Liner

-le

- This is actually two options
  - -l => all print statements should automatically have a newline appended to them
  - -e => tells the perl executable to compile and execute the next item (within quotes)

Our first One-Liner

'print q{Hello World}'

- The perl 'script' we want to compile and execute, in this case the statement *print q{Hello World}*
- Note: We must use the q{} form of quoting, as the next ' seen will close the statement we would want to execute

Extending the One-Liner

- You can put more in the one liner. We can use modules, and do multiple statements, including loops

  >perl -MFile::Slurp -le '@lines = read_file(q{data/text}); foreach my $line (@lines) { print $line; }; print q{read file data/text};'
- Lets look at the new parts

Extending the One-Liner

-MFile::Slurp

- This is equivalent to the *use* statement. We can have as many of these as we need

  -MFile::Slurp -MModern::Perl -MTest::More...

Extending the One-Liner

'@lines = read_file(q{data/text}); foreach my $line (@lines) { print $line; }; print q{read file data/text};'

- How ugly is that. Anyone want a try to read what we are doing?

Extending the One-Liner

'@lines = read_file(q{data/text}); foreach my $line (@lines) { print $line; }; print q{read file data/text};'

- We have 3 statements here, all chained, since perl is whitespace agnostic (i.e. it doesn't matter how many whitespaces there are, and the newline isn't needed at the end of a statement), we can just write statement (or block) after statement, as long as we include the statement separator.

Extending the One-Liner

In a script, we'd have

```
@lines = read_file( q{data/text} );
foreach my $line ( @lines ) {
  print $line;
}
print q{read file data/text};
```

- Much more readable.
- Anyway, let's run it!

Extending the One-Liner

```
This is some

text. If you want my advice, don't

read me with a one-liner
read file data/text
```

- We can write an entire script in the one-liner, the limit is that you can't have more characters than the os will allow.

Extending the One-Liner

-n

- This is perhaps one of the most useful 'extras' for a command line.
- It wraps a while (readline) block around your code, automatically reading out of the file given as an argument at the end (@ARGV)

Extending the One-Liner

- Let's try it

  perl -nle 'print;' data/text
- This is obviously a great way of doing some simple processing, and we can stick as many files on as we like

  perl -nle 'print;' data/text data/one data/two

Extending the One-Liner

- One more worth looking at

  -i
- This modifies a file in place, so we can change things

  perl -lni -e 's/a/i/img; print;' data/for_modifying
- Or with a backup file created

  perl -lni.bak -e 's/a/i/img; print;' data/for_modifying

Extending the One-Liner

- You can also extend by piping in or out of the one liner, for example, we only want to grep for lines that contain read

  >perl -nle 'print;' data/text data/one data/two | grep read

  read me with a one-liner

When to Use

- Because of the ability to pipe from/into other commands, one liners get significant power
- You can use the power of perl to manipulate outcomes, but the power of other programs like cat/grep save you needing to reinvent the wheel

When to Use

- Lets look for something which could find all files with a size less than 72 bytes in the data

  ls -l data/ | grep -v total | perl -le '@lines = <STDIN>; @capture; foreach $file_info (@lines) { @temp = split /\s+/xms, $file_info; if ($temp[4] < 72) { push @capture, $temp[-1]; } }; print join "\n", @capture;'

- pipes into the perl gives access to the data via STDIN
- So only use when the power of perl helps you find something

When Not to Use

- Because you have the full power of perl, it can be tempting to just write a one-liner to solve a problem.
- I mean, if
  - you are only going to want to solve this once,
  - or if you just want to see if something is stuck,

  then surely it's not necessary to write a script.

When Not to Use

Wrong!

When Not to Use

- If you are writing more than 3 statements in a one-liner, think about why you are writing this.
- If you are writing the one-liner to tell you something about a pipeline you are running, don't you think you'll want to run it tomorrow, or next week?

When Not to Use

- If you expect to run it more than once, you should think about writing a script:
  - 1) Only call the script name, even if you do put it in pipes
  - 2) You can write some tests for it
  - 3) Someone will ask you for it
  - 4) It will become part of a project
  - 5) You can start to refactor and capture the code

How do I, Andy, Use Them

- You can probably tell I'm not overly keen on one-liners.
- I use them solely for testing out something I can't remember what will happen.

How do I, Andy, Use Them

- i.e. If I want check if I *print* the array or scalar with *print @array*
- My code I want to write is

  ...time consuming code which generates an array...

  print @array;

- I don't want to run my script to check what I want it to print, but I can't remember if I need the scalar keyword to display the number of elements with print.

How do I, Andy, Use Them

- So, on the command line, I quickly write

  >perl -le '@array = qw{hello goodbye smith jones}; print @array'

  hellogoodbyesmithjones

- So print just concatenates all the array together and prints it, so I quickly tap up, and add in scalar

  >perl -le '@array = qw{hello goodbye smith jones}; print scalar @array'

  4

How do I, Andy, Use Them

- So, I know that in order to print the number of elements, I need to remember to explicitly write in scalar
- Job done. I can move on with my code. Everything else I do, I write in a script/module/object and test! I know I am going to want to do it more than once.
- Even if I do only end up using the script once, it is practice.

References and Extended Data Structures

Aims

- Gain a grasp of references
- Multi-dimensional data structures

2 Dimensional Data

- Often we are going to have rows/sets of data, rather than just a single set
- The current variables we have only cope with one set
  - single value – scalar
  - list – array
  - named data - hash
- How do we deal with a table of data for example?

Data Structures

Name	Age	Height	Hair	Eyes
Andy	36	175	Brown	Blue
James	42	180	Black	Brown
Cameron	5	120	Blonde	Blue

- Here is a table of information
- You are limited to AN array or A hash per row with what we currently know

References

- However, we can use references to help with this problem
- A reference is a pointer to the memory location of the data
- It is stored in a scalar, and we access by dereferencing

References - Scalar

- bin/01-references.pl

```
my $original_scalar = 'original';
my $copy_of_original_scalar = $original_scalar;
say $original_scalar . ' ' . \$original_scalar;
say $copy_of_original_scalar . ' ' . \
$copy_of_original_scalar;
```

- Both of these pieces of information use different memory locations, even though one is a copy.
- \ tells perl to give us the memory reference

References - Scalar

```
$copy_of_original_scalar = 'new';
say $copy_of_original_scalar . ' ' . \
$copy_of_original_scalar;
```

- What you see here is that we have overwritten the value in the copy, but the memory location is the same

References - Scalar

```
my $reference_to_original_scalar = \
$original_scalar;
say $reference_to_original_scalar . ' ' . \
$original_scalar;
```

- We can assign the reference to another variable

References - Scalar

- To get it back:

```
say ${ $reference_to_original_scalar };
```

- By putting it into ${ } we tell perl to dereference the reference to a scalar

References - Scalar

- Now a bit of Magic:

```
$original_scalar = 'another string';
```

- What do you expect to see if we get the dereferenced value?

```
say $original_scalar . ' ' . $
{ $reference_to_original_scalar };
```

References - Scalar

- And what about changing the value of the dereferenced reference?

```
${ $reference_to_original_scalar } = 'original
again';
say $original_scalar . ' ' . $
{ $reference_to_original_scalar }
```

References - Scalar

- Why faff with references to scalars?
  - Can reduce memory (do you want two copies of War and Peace?)
- What we see is the principal. We can take references of any data structure.

References - Array

```
my @original_array = 1..200;
my @copy_of_original_array = @original_array;
say \@original_array;
say \@copy_of_original_array;
```

- Here we are, doing the copy of an array again, and seeing the memory locations of the two copies

References - Array

- Lets take a reference of the original array

```
my $array_ref = \@original_array;
```

- Note: We are storing the reference in a scalar, not an array!

```
say $array_ref . ' ' . \@original_array;
```

- Same reference to place in memory
- We can get an element with the - > operator

```
say $array_ref->[0];
```

References - Array

- We can do similar magic:

```
splice @copy_of_original_array, 100,10;
say scalar @copy_of_original_array;
say scalar @original_array;
say scalar @{ $array_ref };
```

- Whats going to be the results here?

(splice is removing 10 elements, starting at index 100, scalar returns the number of elements)

References - Array

```
push @original_array, @original_array;
say scalar @copy_of_original_array;
say scalar @original_array;
say scalar @{ $array_ref };
```

- How about here?

(This is a way of doubling an array)

References - Array

```
pop @{ $array_ref };
say scalar @copy_of_original_array;
say scalar @original_array;
say scalar @{ $array_ref };
```

- And here?

(pop removes the last element)

References - Array

- Basically, we have only altered the array stored in memory
- You can do anything you want with an arrayref that you can do with an array
- You just need to dereference with
  - @{ $array_ref } for the full array
  - $array_ref->[index] for individual element

References - Hash

```
my %original_hash = (
  band => 'Queen',
  'lead vocals' => 'Freddie Mercury',
  'lead guitar' => 'Brian May',
  'bass guitar' => 'John Deacon',
  'drums' => 'Roger Taylor',
);
```

- Here I have a hash, anyone want to suggest how I take a reference?

References - Hash

```
my %copy_of_original_hash = %original_hash;
say \%original_hash;
say \%copy_of_original_hash;
my $hash_ref = \%original_hash;
say $hash_ref . ' ' . \%original_hash; # same reference to place in memory
```

- Again - same reference to place in memory

References - Hash

```
$copy_of_original_hash{band} = 'Queen + Paul Rodgers';
$copy_of_original_hash{'lead vocals'} = 'Paul Rodgers';
delete $copy_of_original_hash{'bass guitar'};
say %original_hash;
say %copy_of_original_hash;
```

- Here, we have modified the copy band

Instantiate Anonymity

- References are often referred to as anonymous, and can be directly created

```
my $anon_array_ref = [ ];
say $anon_array_ref;
my $anon_hash_ref = { };
say $anon_hash_ref;
```

- This will be very important for later work

Multi-Dimensional Data

- So an array or hash can be referenced and that can be stored in a scalar 'box'
- As arrays or hashes contain scalar boxes, we can put references in those boxes as well

bin/02-multi_dimensional_data_structures.pl

Multi-Dimensional Data

```
my %amino_acid_3_letter_codes = (
  Alanine => 'Ala',
  Arginine => 'Arg', ...
);
my %amino_acid_bases = (
  TTT => 'Phenylalanine',
  TCT => 'Serine', ...
);
```

Multi-Dimensional Data

- The first thing to note is that there is lots of repeated data inside the 2nd hash
- But, we could change that around, so that the codons are in an array, which match the name, using arrayrefs

Multi-Dimensional Data

```
my %codes_per_amino_acid = (
  Alanine => [ 'GCA', 'GCC', 'GCG','GCT],
  Arginine => [ 'AGA', 'AGG', 'CGA', 'CGC', 'CGG', 'CGT' ], ...
);
```

- we are instantiating an anonymous array into the scalar that would be at $codes_per_amino_acid{Alanine}

Multi-Dimensional Data

- Could we get the two hashes down to one?

  Yes
- Because hashes can also be referenced.

```
my %genetic_code_data = (
  'Alanine' => {
    '3_letter_code' => 'Ala',
    'codons' => [ 'GCA', 'GCC', 'GCG', 'GCT' ]
  }, ...
);
```

Multi-Dimensional Data

- How do we access the data in a multidimensional structure?

```
say $genetic_code_data{Methionine}{'3_letter_code'};
say $genetic_code_data{Methionine}{codons}[0];
```

- We chain together the keys/indexes that would be used in each structure to get the final value.

Multi-Dimensional Data

- We can just have arrays of course

```
my @multiplication_table = (
  [ qw{ 0 0 0 0 0 0 0 0 0 0 0 } ],
  [ 0..10 ],
  [ qw{ 0 2 4 6 8 10 12 14 16 18 20 } ],
  [ qw{ 0 3 6 9 12 15 18 21 24 27 30 } ],
);
say $multiplication_table[3][5];
```

Summary

- By using references, we can store multidimensional data (tables)
- A scalar can contain a reference to a memory location of an array or hash
- Each of these items can also contain memory references to further arrays and hashes
- We can retrieve the data using a combination of indices and keys, and the ->

Functions 2

Aims

- Recap what a function is / does / benefit
- How to write a function
  - How to give it inputs
  - How to use inputs
  - What to get as an output

The rest of the course

- At this point, I'd like to point out how we are going to take the code with the rest of the course.
- We are going to use Sequence Manipulation as our project, and move from
  - a procedural script, (what we have done)
  - To scripts with functions, (this section)
  - To making those function reusable between scripts with modules
  - To then turn modules to objects

What is a Function?

- Functions do something for us with some inputs
- They can be part of the core Perl, or be something we imported (like write_file from File: :Slurp)
- They can save us a lot of coding

Function - Syntax

- We have seen the following

```
if ( $true ) {
  ...do something
}
print 'true' if $true;
say 'Hello World';
my $data = read_file( 'data_file' );
```

Function – Syntax - Parameters

- Parameters are always a list (array) of items after the function name
  - if ($true) - the braces indicate the lit of items to pass to if
  - if $true - there is a single item, but this is passed to if as the first item of a list
  - print 'true' - we pass a single string as the first item of a list to print
  - say 'Hello World' - we pass a single string as the first item of a list to say
  - read_file('data_file') - we pass a list of items, containing 'data_file' to read_file
- Functions require a list of parameters.

Function – Syntax - Return

- Functions must return to the calling code
- With a return value which we can capture
  - *my $data = read_file('data_file');*
- We can do something if it is true
  - *print 'true' if $true;*
- If the return value of the call to *if* is true, then we will *print*

Function – Syntax - Return

- We can execute a block of code
  - *while ($count < 10) {*
    say 'below 10';
    }
- In this case, perl checks the return value for us
- We could also ignore the return value
  - *say 'Hello World';*

Function – Syntax

- Since we know the way functions are called, let's create a function that takes parameters and returns something.

Write a Function – Syntax

- The syntax for writing a function is

```
sub function_name {
  my @params = @_;
  ...do something
  return <values>;
}
```

- Let's take a look at this

Write a Function – Syntax

sub function_name {

- This is actually passing two parameters (a function name and a block of code) to the function *sub*. This generates a new function with your name, which when called, will run the code in the block.

Write a Function – Syntax

my @params = @_;

- The list of parameters you give to a function are passed in as an array called @_
- Very bad to use this to process directly, so assign this to a new array (make a copy of it), and scoped to the function code block.
- Note: 'scoped', only within the block of code that it has been created within

Write a Function – Syntax

...do something

- Run some perl code within the block

return <values>;

- We want to return to the caller, and pass back any values we have created (as the scope will stop them being visible outside of the function). We can pass back either undef (nothing), a scalar or an array.

Write a Function – Syntax

}

- Close the block of code (or else at best we will fail to compile)
- OK, now let's create a function called *find_methionine* which takes a sequence of DNA, and returns true if the Methionine codon (ATG) is found in it

Function – *find_methionine*

- From the functions we have seen so far, what might we want to use to look for methionine?
- What true and false values might be used?

bin/02-first_function.pl

Function – *find_methionine*

sub find_methionine {

- First, lets declare our function, and start a code block

my ($sequence) = @_;

- Now, we get the sequence from the variables passed to the code block

my $return;

- set up a return variable, preset to a false value

Function – *find_methionine*

```
if ( $sequence =~ m/ATG/ixms ) {
  $return = 1;
}
```

- our bit of code that we want to run, in this case a pattern match against the methionine codon, setting *$return* to true if we match

Function – *find_methionine*

return *$return;*

- Return the *$return* variable, which will now be true if matched, or false if it didn't

}

- Close our code block

Function – *find_methionine*

- Let's try it out. Since we know it should return true or false, we'll stick it in an *if/else* block

```
my $sequence =
'ACTGATCGATAGCTAGCTAGCTAGGCT';
if ( find_methionine( $sequence ) ) {
  say "$sequence contains Methionine";
} else {
  say "$sequence does not contain Methionine"
}
```

Function – *find_methionine*

- That should not have found it, let add ATG

```
$sequence .= 'ATG';
if ( find_methionine( $sequence ) ) {
  say "$sequence contains Methionine";
} else {
  say "$sequence does not contain Methionine"
}
```

- The script has an array of sequences, and the test in a loop. Take a look.

Function – *find_methionine*

- What was the benefit? I could do a pattern match against each one.
- Yes, but can you already see that instead of writing

```
if ( $sequence =~ m/ATG/ixms ) {
```

- three times, I have written

```
if ( find_methionine( $sequence ) ) {
```

three times - less chance of putting the wrong codon in, or match options.

Function – *find_methionine*

- Imagine if there were more code needed here as well, or we wanted to extend the code.
- You would only need to change in one place.
- Let's extend.

bin/03-extend_function.pl

find_amino_acid_by_codon

- There are 20 amino acid bases, plus stop to think about. 64 codon combinations. That is between 21 and 64 lots of almost the same code repeated.
- We can pass in any number of parameters, not just the sequence. So, why not pass in the codon or amino acid we want to find as well.

find_amino_acid_by_codon

```
sub find_amino_acid_by_codon {
  my ( $sequence, $codon ) = @_; # 2nd parameter
  my $return;
  if ( $sequence =~ m/$codon/ixms ) { # match the
$codon variable, rather than a fixed string
    $return = 1;
  }
  return $return;
}
```

find_amino_acid_by_codon

```
my @seqs = qw{...};
foreach my $seq ( @seqs ) {
  if ( find_amino_acid_by_codon( $seq, 'AAA' ) ) {
    say "$seq contains Lysine";
  } else {
    say "$seq does not contain Lysine"
  }
}
```

find_amino_acid_by_codon

- So, now we have a function which means we never have to write the pattern match again.
- However, we need a source of codons to look at, and we don't want to have to remember there are 2 for Lysine, 3 for STOP...

find_amino_acid

- Lets try and extend the function so that we can see if the amino acid is present by it's name, checking all codons relating to it.

bin/04-extend_function_again.pl

find_amino_acid

```
sub find_amino_acid {
```

- we change the name to be more generic

```
  my ( $sequence, $amino_acid ) = @_;
```

- and do the same for the variable, as *$codon* suggests a sequence

```
  say "I am going to see if I can find $amino_acid in $sequence:";
```

- Lets move part of the output message - what it is doing - into the function

find_amino_acid

```
  my %genetic_code_data = (
    'Alanine' => [ 'GCA', 'GCC', 'GCG', 'GCT' ],
    'Lysine' => [ 'AAA', 'AAG' ],
    'Methionine' => [ 'ATG' ],
    'Stop' => [ 'TAA', 'TAG', 'TGA' ],
  );
```

- Lets store within the function the amino acids and the codons associated with them. Could be a file/db call.

find_amino_acid

```
  my $return;
```

- We want our return 'flag';

```
  foreach my $codon
  ( @{ $genetic_code_data{ $amino_acid } } ) {
```

- We'll loop over the array of codon sequences found for the requested amino acid

find_amino_acid

```
    if ( $sequence =~ m/$codon/ixms ) {
      $return = 1;
      last;
    }
```

- and do our match - we only care if we see a match, so lets get out (*last*) as soon as we match to improve speed

find_amino_acid

```
  }
  return $return;
}
```

- Close, the loop block, return our answer, and close function block
- Let's see it working

find_amino_acid

```
my @seqs = qw{...};
foreach my $amino_acid ( qw{Alanine Lysine Methionine Stop} ) {
```

- Loop over an array of amino acid names

```
  foreach my $sequence ( @seqs ) {
```

- Loop over each of the sequences

find_amino_acid

```
    if ( find_amino_acid( $sequence, $amino_acid ) ) {
```

- Make a call to our extended, more generic function, which takes the amino acid we are currently looking at.

find_amino_acid

```
      say "\tFound";
    } else {
      say "\tNot Found";
    }
```

- we only want to say found or not found here, since the function will report sequence and amino acid

```
  }
}
```

Functions - Summary

- Functions help us to stop writing the same thing over and over again
- We can use them to simplify the code
- We can extend them to do more
- Functions can call other functions, and within them you can do anything that you could do elsewhere in Perl
- Think of them like a mini-script, returning values to the main script

Functions - Workshop

- 08a-write_a_function directory
- bin/01-write_a_script_with_functions.pl
- data/amino_acid_codes.txt
- data/sequences.txt
- output/sequences_with_codons_present.txt
- create a script (using functions) to create a file which, given a file of sequences and a file of amino acids, lists the amino acids next to the sequence i.e. ACGTAGCT.. Ala Stp..

Modules

Aims

- Understand what a module is
- Understand why they are useful
- Be able to create a module
- Know that they are usually kept in a lib directory
- Know that you can get modules from the CPAN

Modules

- So far we have created scripts with functions and seen functions be called
- At the end of our last workshop, we had a script with many functions, that could be useful in other scripts
- How usefult is it to see if a codon is present? How many scripts could use that?

Modules

- Problem: if the code is in a script, only the script has access to it.
- Solution: Put that code into some code which can be used by may scripts – A Module.

Modules – They are scary?

- 3 reasons I have heard why modules are scary
  - 1) Outside of my script, I can see what is going in in my script
  - 2) Is it really perl?
  - 3) Extra files are likely to get it noticed

Modules – They are scary?

- 1) Outside of my script, I can see what is going in in my script
  - Do you really want that much cut and pasting? If the function name is good, you can see what is happening
- 2) Is it really perl?
  - Yes
- 3) Extra files are likely to get it noticed
  - This is a good thing(tm), support long term

Modules – They are scary?

- So, don't be scared. You are only writing some code in another file, to enable it to be reused.

What is a Module?

- Firstly, it is a file of perl code, named with a .pm file extension instead of .pl
- You often find them in a directory called lib
- Module code contains a number of things. Some optional, some not.

What is in a Module?

- 1) A *package* name

  package DnaHelpers;
- Perl code can be grouped into a package. Essentially, this is the code inside the module.
- Compulsory, as Perl uses this, and must be the first line of code in your file.
- The name of the file, less the .pm extension

What is in a Module?

- This enables our script to

  use DnaHelpers;
- Which we have seen before with

  use File::Slurp;

What is in a Module?

- 2) functions
- We can write any functions we like in here.
- They do not need to
  - know about one another,
  - use the same variables,
  - have any true bearing on the theme of the module (although you should have a reason for putting extra stuff in).

What is in a Module?

- The functions are written in precisely the same way as they are in a script.

```
sub find_amino_acid {
  my ( $sequence, $amino_acid ) = @_;
  ...
  return $return;
}
```

What is in a Module?

- 3) A way of exporting those functions
- There are two ways of utilising functions in a Module. We are going to look at the non-object oriented way in this section.

```
use base 'Exporter';
our @EXPORT = qw{find_amino_acid};
```

What is in a Module?

- First we use another module to allow us to export methods

  use base 'Exporter';
- All the functions listed in this special array

  our @EXPORT = qw{find_amino_acid};
- will now be handed over to the script that calls

  use DnaHelpers;
- and this script can now *find_amino_acid()*

What is in a Module?

- 4) Call in other modules

  use Modern::Perl;
- As the module is just perl code, it can call in other modules to use just like a script.
- And therefore get access to all the methods they export

What is in a Module?

- 5) No code outside of the functions, apart from setup
- Modules are not scripts, so writing

  say 'Hello World';
- outside a function will confuse perl.

What is in a Module?

- 6) 1;
- It is compulsory for the last line of your module to be a true statement. Therefore, most (all?) modules will finish with

  1;
- in order to ensure that this requirement is fulfilled.

How to use a Module

- Now, in order to use the module, we need to tell the script where it can be found

  use lib 'lib';
- This tells our script to also look in the lib directory of the current directory for modules

  use DnaHelpers;
- The script now knows to look for a file DnaHelpers.pm, and to load in the code

How to use a Module

- We have taken the clean script from the last workshop and called it

  bin/01-many_function_script.pl
- What we want to do is take a function out of here, and move it to a module. Let's take out *find_amino_acid*

  lib/DnaHelpers.pm

  bin/01-without_find_amino_acid.pl

How to use a Module

lib/DnaHelpers.pm

- This exports our *find_amino_acid* function, and has it present

  bin/01-without_find_amino_acid.pl
- This script has removed the *find_amino_acid* function, and has '*use*'d DnaHelpers.pm
- Run the script. It works as before.

Extend the Module

- Of course, we have more functions still in our script that we'd like to use elsewhere.
- Such as retrieving the *amino_acid_codes*, as there is no point rewriting the code to obtain the data from the file,
- and even *sequence_and_present_amino_acid_codes* is likely to prove useful elsewhere
- So lets try to take them out

Extend the Module

lib/DnaHelpersExtended.pm

- Before we move a function across, take a look at the other functions it is using
- *amino_acid_codes* uses read_file, so we need to

  use File ::Slurp;
- But apart from that, we can move it from the script, and add it to the @EXPORT array

Extend the Module

- We won't move across

  write_sequences_with_amino_acid_codes_to_file
- as this is specific to how we want to act on the data, and probably won't be useful to other scripts

Extend the Module

- So now look at *bin/01-without_many.pl*
- The script has dramatically less code, but does exactly what we want it to
- lib/DnaHelpersExtended.pm has code for others to use

CPAN

- Most (all worthwhile) reusable code is found in modules.
- The greatest source of these are found on the Comprehensive Perl Archive Network (CPAN).
- Chances are, if you need to do it, someone has already written a module to do it.
- Everything on the CPAN is free for you to download and use in your own projects.

CPAN

- It is also worth noting that there are extensive bioperl tools there, so most of what we are doing in this course is actually re-inventing the wheel.
- Search for it on CPAN, and then find it and use it, or write it and submit it back.

Modules - Summary

- A module is a file of reusable code
- It contains functions that can be used in many scripts
- There are some compulsory parts to it
  - Package name
  - True end value (1)
  - Export array
- Usually kept in a lib directory

Modules - Workshop

- cd workshop
- Two scripts to work on this time
  - 01-write_module.pl
  - 02-reuse_module.pl
- You will need to refer back to stuff we have done before to have your functions do what is requested

Objects

Aims

- Understand what a object is
- Understand why they are useful
  - Data with the code which processes it
- Know that they are usually kept in a lib directory
- Format much like modules
- Moose is best!

Procedural Processing

- Perl originally conceived as a procedural language.
- Execute a script, top to bottom. Much like following the recipe for making a cup of tea.

Procedural Processing

- Procedure
  - Put teabag in cup
  - boil water
  - add boiling water to cup
  - brew to taste
  - remove teabag
  - add milk/sugar/lemon to taste
  - drink

Procedural Processing

- Procedural processing is fast
- Because computers (like us) like to process in a top to bottom way
- You don't start a recipe halfway down - or at least not more than once :)

Object Oriented Processing

- Many other languages are considered Object Oriented
- They like everything to be an object, and you call methods (functions) on that object.
- Much like a kettle is an object, it can't make the tea, but it can boil water.

Object Oriented Processing

- Cup
  - contains water, teabag, milk, sugar, lemon (attributes)
  - has methods add_water, add_teabag, add_milk, add_sugar, add_lemon
- Kettle
  - contains water
  - has methods boil_contents, fill_kettle, pour_contents, water_is_cold, water_is_hot

Object Oriented Processing

- Saucepan
  - contains water, soup
  - has methods boil_contents, fill_saucepan_with, simmer_soup, pour_contents
- Teabag
  - has methods fresh, is_used

Object Oriented Perl

- In Perl5, the designers decided to implement an Object system into Perl.
- A Perl project could become object oriented if the developer of that project wished it.
- What actually happened was that Perl virtually became Object Oriented.
- Procedural scripts to be only when objects were really to much.

Object Oriented Perl

- Object Orientation is an add on, it is slower than procedural.
- But has some nice advantages, such as allowing us to 'abstract' away code which can obtain 'data' in different ways.
- Does our teamaker need to worry about where it gets boiled water? It could use a kettle or a saucepan.
- Objects represent this in the code.

Object Frameworks

- Frameworks developed to take advantage of this, and a framework has appeared which does 90% of the hard work for you in creating objects.
- Moose.

Object Creation

- We are going to create objects, and use them in our scripts, using Moose.
- It is by far the easiest way to create objects, leaving out
  - how to set up the objects
  - how to (correctly) store data in them.
- We are going to keep our objects fairly simple.

Our first Object - SequenceManipulator

- First and foremost, an object is a package, more usually referred to as a class.
- We write a module (*lib/SequenceManipulator.pm*)

  package SequenceManipulator;
- a package name (compulsory)

  1;
- the final true value (compulsory)

Our first Object - SequenceManipulator

- Perl has no idea that this is an object, nor will it until it has been instantiated as one.
- For this, we need to provide a method called *new*.
- Thankfully, this is where Moose comes in.

  package SequenceManipulator;

  use Modern::Perl;

  use Moose;

  1;

Our first Object - SequenceManipulator

- The *new* method is virtually the same in all objects you could write, so Moose stops us copy and pasting that method.
- Note, we don't need any of that export stuff. Objects keep all their functions or 'methods' with them.
- Let's try using the object in a script.

  bin/01-use_SequenceManipulator.pl

Our first Object - SequenceManipulator

use lib 'lib';

- we still need to tell the script where to look for our class module

  use SequenceManipulator;
- We need to 'use' the module, for the interpreter to compile the code for us

Our first Object - SequenceManipulator

my $sequence_manipulator = SequenceManipulator->new();

- Create the object
- Moose gives automatically gives our class module the method new, which we call using the arrow (->) operator.
- Same principal as using this for key-values or index-values in reference data structures.

Our first Object - SequenceManipulator

my $sequence_manipulator = SequenceManipulator->new();

- Create the object
- Moose gives automatically gives our class module the method new, which we call using the arrow (->) operator.
- Same principal as using this for key-values or index-values in reference data structures.

Our first Object - SequenceManipulator

- If we take a look at the variable *$sequence_manipulator*

```
say $sequence_manipulator; #
SequenceManipulator=HASH(0x8100e0)
```

- Lets look at this

Our first Object - SequenceManipulator

HASH(0x8100e0)

- The first thing we spot - we have a hash reference. Just a like we saw in section 3 - variables and data structures – references
- So we have a data structure reference of the type HASH.
- What does the other part mean though?

Our first Object - SequenceManipulator

SequenceManipulator=

- We see our package or class name, with the = sign.
- The HASH reference has been *blessed* into the package SequenceManipulator.
- Don't worry about the term *blessed*. Basically we have an object of the class *SequenceManipulator*, which we will be able to call methods and attributes on.

Objects – Add attributes

- When we discussed the cup as an object, we said it had attributes such as
  - water
  - teabag
  - milk
  - sugar
  - lemon
- Does the cup have water in it? Yes/No

Objects – Add attributes

- Attributes are just the data contents of an object
- We make things easy by having attributes relevant to the object
  - the object is going to do something with them (change water to tea)
  - logically provides a store for them (library stores books, but doesn't store kettles)

Objects – Add attributes

- Our SequenceManipulator object will need some attributes. Suggestions?

Objects – Add attributes

- Our SequenceManipulator object will need some attributes. Suggestions?
  - Sequence
- It is logical for a SequenceManipulator object to have a sequence attribute, so that it has something to manipulate

lib/SequenceManipulatorAttributes.pm

Objects – Add attributes

```
has 'sequence' => (
  is => 'rw',
);
```

- *has* is a special method which Moose gives us, to create an attribute with the name provided, in this case '*sequence*'.
- Need an attribute definition hash.
- You must always provide an 'is' definition.
- Two values rw or ro.

Objects – Add attributes

- We can also define what type of data can be stored
- As sequence must be a string, we will define that

```
has 'sequence' => (
  isa => 'Str',
  is => 'rw',
);
```

- We won't go into any more definitions.

Objects – Add attributes

bin/02-adding_attributes.pl

```
my $sequence_manipulator =
SequenceManipulatorAttributes->new( {
  sequence => 'ACTAGTC...AGTCGTAT',
} );
```

- We add the sequence to the attribute when we create the object with a hashref with keys that match the attribute names, and values for those keys.

Objects – Add attributes

bin/02-adding_attributes.pl

```
say $sequence_manipulator;
say $sequence_manipulator->sequence();
```

- To call back the sequence, all we need to do is call the attribute name off the object via the arrow operator.

Objects – Add attributes

- We can also add a sequence to the object after it has been created (but only if using rw)

```
my $sequence_manipulator_2 =
SequenceManipulatorAttributes->new();
say $sequence_manipulator_2;
$sequence_manipulator_2->sequence( 'agcgatgct' );
say $sequence_manipulator_2->sequence();
```

- We pass in the sequence to the attribute name, again called via the arrow operator

Objects – Add attributes

- We can have more than 1 attribute
- You will do this in the next workshop.

Objects – I could use a hash

- I could just get a hash, store a sequence on that as a key, and call

```
my %sequencehash = (
  sequence => 'AGCTGATG',
);
say $sequencehash{sequence}
$sequencehash{sequence} = 'gatgtagt';
```

- True, but object have methods

Objects – Add Methods

- Object have methods.
- Our kettle object had methods
  - boil_contents
  - fill_kettle
  - pour_contents
  - water_is_cold
  - water_is_hot

Objects – Add Methods

- What we can see is that the methods this object has are related to the attributes it would have
  - contains_water
- This is what we get with objects. Data, with methods that act on that data.

Objects – Add Methods

- So for our SequenceManipulator, what methods might we have?

Objects – Add Methods

- So for our SequenceManipulator, what methods might we have?
  - length
  - at_content, gc_content
  - at_percentage, gc_percentage
  - find_amino_acid
  - reverse_strand
  - amino_acid_sequences_per_orf
  - change_base

Objects – Add Methods

- We have seen some of these before.
- Why might it be better to keep them in the object, with the sequence, rather than outside?
  - e.g. just exported from another module - after all, being in a module makes it reusable

Objects – Add Methods

- If you keep your data and the methods which manipulate it close, the code becomes clearer.
- *length* could mean anything
  - part of the dimensions of a piece of office furniture
  - requested length for a table to be
  - time portion

Objects – Add Methods

- If the method is inside the object, then it limits the scope that length can mean.
- It has to relate to sequence, and therefore is likely to be either
  - a request for a specific length,
  - or more likely, the length of the sequence in the manipulator.

Objects – Add Methods

- Reduce the likelihood of clashes of code, since you have to call the method off the object so that

```
$table->length();
$sequence_manipulator->length();
```

- will know which operations to do on which data, rather than

```
length( $table );
length( $sequence );
```

Objects – Add Methods

- So, let's add a method to our SequenceManipulator.

```
sub length { };
```

- That was easy, wasn't it. All we have done is written a function in our module.
- However, it won't allow us to manipulate the sequence, as it hasn't got a clue about the object.

Objects – Add Methods

```
sub length {
  my ( $self ) = @_;
};
```

- First parameter passed to an objects method is the object itself.
- Always assign to the *$self* variable!
- Now the method can access the attributes on the object with

```
$self->attribute_name();
```

Objects – Add Methods

- So let's finish our length method

```
sub length {
  my ( $self ) = @_;
  return length $self->sequence();
}
```

lib/SequenceManipulatorMethods.pm

bin/03-adding_methods.pl

- We can call the length on the sequence

```
say $sequence_manipulator->length();
```

Objects – Add Methods

- Lets add the ability to change a base in the sequence to another one, permanently.

```
sub change_base {
  my ( $self, $original_base, $new_base ) = @_;
  if ( $original_base !~ m/[ACGTN]/ixms || $new_base
!~ m/[ACGTN]/ixms ) {
    die 'one of your bases is not a legitimate base';
  }
```

Objects – Add Methods

```
  my $sequence = $self->sequence();
  $sequence =~ s/$original_base/$new_base/gixms;
  $self->sequence( $sequence );

  return $sequence;
}
```

bin/03-adding_methods2.pl

```
$sequence_manipulator->change_base( 'a', 't' );
say $sequence_manipulator->sequence();
```

Objects – Add Methods

- This is more complicated, let's go through it

```
sub change_base {
  my ( $self, $original_base, $new_base ) = @_;
```

- We want to accept two parameters. $original_base and $new_base.
- However, the first parameter to an object method is always itself, so we need to account for this, and accept 3 parameters, naming accordingly.

Objects – Add Methods

```
  if ( $original_base !~ m/[ACGTN]/ixms || $new_base
!~ m/[ACGTN]/ixms ) {
    die 'one of your bases is not a legitimate base';
  }
```

- We'll do a check here to ensure that the 2 parameters match a legitimate DNA letter (we'll include N).
- If not, we'll *die*, which will take us back to the calling script and *die* out of that with the message declared.

Objects – Add Methods

```
  my $sequence = $self->sequence();
```

- We'll get the sequence that is inside the object.
- We have to do that as a call on $self, since there is no magic which will let us use the attribute without such a call.

```
  $sequence =~ s/$original_base/$new_base/gixms;
```

- We do a regular expression substitution on the sequence obtained.

Objects – Add Methods

```
  $self->sequence( $sequence );
```

- We store this back in the attribute (which is why we need the attribute to be read/write)

```
  return $sequence;
}
```

- We return the changed sequence, as this provides a true value, and close the method.

Objects – Add Methods

```
$sequence_manipulator->change_base( 'a', 't' );
```

- We call the method on object *$sequence_manipulator*, passing in 2 parameters
  - the base we want to change
  - the base we want it changed to

Objects – Please Sir, Can I have some more?

- We can add any amount of methods we like, and any number of attributes.
- However, it is recommended that you look to keep objects discrete, and methods relevant.
- It's no use creating an object that can do loads of stuff, if the call for it to do it is highly unlikely.

Objects – Please Sir, Can I have some more?

- i.e. Just to see if *Gattaca* is found

```
sub find_film_name {
  my ( $self, $film_name ) = @_;
  if ( $self->sequence() =~ m/$film_name/ixms ) {
    say "film $film_name found.";
  } else {
    say "film $film_name not found.";
  }
  return;
}
```

Objects – Summary

- 1) Almost identical to a module
  - with a package name and the true 1;
  - But use Moose, instead of Export stuff

```
package MyObject;
use Modern::Perl;
use Moose;
1;
```

Objects – Summary

- 2) You still use the package in your script

```
use MyObject;
```

Objects – Summary

- 3) An object has attributes - data stores which can be read/write or read only, and can be typed
  - has 'string' => (isa => 'Str',is => 'rw');
  - has 'num' => (isa => 'Num',is => 'ro');
- If it needs modifying either outside of the object or within after construction, it need to be rw

Objects – Summary

- 4) An object has methods - written almost identically as functions, except needing to get *$self*

```
sub my_object_method {
  my ( $self, $param1, $param2 ... ) = @_;
}
```

Objects – Summary

- If you want to access an attribute to work on, then

```
my $method_copy = $self->attribute();
```

- Once worked on, if it has to be overwritten or populated (and is rw)

```
$self->attribute( $worked_on_copy );
```

Objects – Summary

- 5) Obtain an object in your script with

```
my $object = MyObject->new( {
  attribute1 => 'value1,
  ...
} );
```

Objects – Summary

- 6) Call attributes and methods on your object

```
my $object_attribute = $object->attribute1();
$object->my_object_method( @parameters );
```

Objects – Workshop

cd workshop

bin/01-write_an_object.pl

- Looking to create 3 objects
  - Sequencing Machine producing sequences
  - Sequence object
  - Object to produce the amino acid codons

Combining Scripts, Modules and Objects

Aims

- Have an idea of when to use
  - scripts only
  - modules
  - Objects
- Have an idea of how they relate to each other

When do I use a each?

- When you want to develop, there is no reason not to only use scripts if that is what you are comfortable with, but let's look at why you want to consider all the three options.

Script?

- The backbone has to be a script in some format.
- Important features:
  - a) Name - make the name relevant
  - b) Keep the code flow readable.
  - c) Use functions instead of copy and paste
  - d) Only keep code which cannot be used elsewhere, in the script
  - e) any interactivity is best kept to the script

Module?

- If you are writing lots of functions, then you should consider breaking these out into a module.
- Important features:
  - a) Name - make the name of the module(s) relevant to the functions they provide.
  - b) Export the functions
  - c) Make the functions as 'black box' as possible - i.e. *find_sequence* rather than *find_codon*

Object?

- Functions need to be very specific? Trying to represent data? then you should consider making an object.
- Important features:
  - a) Name - it represent the object
  - b) Methods should be relevant to the object
  - c) Make your attributes relevant – i.e. Sequencer attributes that might be relevant, *flowcell, camera, reagents* rather than *contents, equipment*

Example – Produce a Cup of Tea

- Script: make_tea
- This is a process. So we want the script to go from start to end
  - 1) get kettle
  - 2) fill kettle with water
  - 3) boil water
  - 4) get cup
  - 5) get teabag
  - 6) put teabag in cup

Example – Produce a Cup of Tea

- Script: make_tea
  - 7) add boiling water to cup
  - 8) allow to brew
  - 9) remove teabag from cup
  - 10) add sugar
  - 11) add milk
  - 12) add lemon
  - 13) drink
- This is what it needs to do.

Example – Produce a Cup of Tea

- Modules: TeaHelper
- Are there any processes which are repeated, and could be put into a module
- adding something to something else?

Example – Produce a Cup of Tea

- Objects: Kettle, Cup, Teabag
- What do each of them have/do
  - Kettle
    - filled with water, needs to boil water,
    - needs to be able to be poured
  - Cup
    - able to have a teabag, water, milk, sugar,
    - able to remove a teabag, be drunk
  - Teabag
    - can be fresh/spent

Example – Produce a Cup of Tea

- So we can see options to utilise all three,
- although in practice, we probably wouldn't write a module just to provide the adding function, unless we could see benefits outside of making a cuppa.

Summary

- This short section was really just to give you an idea of what you want to think about when developing a new perl program.
- There are plenty of further options, and other things you can do.
- Perls motto of 'There is more than one way to do it' is as applicable here, as when actually writing your code.

Summary

- Think about what you want to achieve first, you can solve almost all of your problems really sensibly.
- What will help you focus more on that though are tests, and that is what we shall now look at.

Testing

Aims

- Know what a test is
- Know why we should test

What Are Tests?

*What is 2*2?*

What is the name of Queen Elizabeth II?

Does the sequence 'ACGAAGTCGAACTAGCTACGAGT' contain the codon for Methionine?

- These are tests. And that is what we are going to look at now.
- How do you test that your code does what you expect it to?

Why Test?

- We have written some scripts, modules and objects and before we release them to the masses, we want to check that they are going to work.
- At least we can then be somewhat confident then that if the users email us saying it doesn't work, we have an idea of why (or at least why not)

How do we Test?

- We could sit there and run the scripts over and over again, but that is
  - going to get tedious
  - will be unreliable
- Can we automate?
- We can. With a module '*Test::More*', and a special program '*prove*' which combined can test our code for us and tell us if it works. It can run this over and over.

A Test Script

- Test generally live in a t/ directory

t/hello_world.pl

```
use Test::More tests => 1;
```

- use the Test::More module, with a tests count of 1, as we want to run one test.

```
my $script_output = qx{bin/hello_world.pl};
```

- wrapping a command in qx{} tells perl to go out to the shell and run that script, returning any STDOUT

A Test Script

```
is(
```

- call the imported 'is' method, with parameters

```
  $script_output,
```

- the item we want to test

```
  "Hello World\n",
```

- the output we expect

A Test Script

```
  'hello_world.pl returns Hello World'
```

- a comment to give against the test in output

```
);
```

- We run the script with prove

```
>prove -v t/hello_world.t
```

- -v means run in verbose mode

A Test Script

```
>prove -v t/hello_world.t
1..1
ok 1 - hello_world.pl returns Hello World
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.03 cusr  0.01 csys =  0.06 CPU)
Result: PASS
```

A Test Script

- Testing a script is useful, but if the script needs you to be interactive, it can be more tricky.
- In these circumstances, you want to be looking at what you need to test, and what is easiest to test.

Module Testing

- Modules are prime to test, because the test is a script, and they expect to be called by a script.
- You test each function in turn, passing in parameters, and testing the expected outcomes.
- Lets test the DnaHelpers module we wrote before.

Module Testing

t/module.t

```
use Test::More tests => 3;
```

- This time there will be 3 tests

```
use_ok ( 'DnaHelpers' );
```

- test that we can use the module ok - this checks it will compile and gets the code for us

Module Testing

```
my $sequence = 'ACATCA..ATGC';
my %amino_acids = (  Methionine => {
   '3_letter_code' => 'Met',
   codons => [ qw{ATG} ]
 }, );
```

- set up some data to use for testing
  - a sample of our Amino Acid data structure,
  - and a sequence that contains ATG

Module Testing

```
is(
 find_amino_acid( $sequence,
$amino_acids{Methionine} ),
 'Met',
 'Methionine found' );
```

- Test to see if the method, with our known data, actually returns the 3 letter code because it finds ATG

Module Testing

```
$sequence =
'ACATCAGATCGTAGCTGCTCGCTGCGATACGC';
is(
 find_amino_acid( $sequence,
$amino_acids{Methionine} ),
 undef,
 'Methionine not found' );
```

- Reset *$sequence* to not have ATG, and test that undef (false) is returned

Module Testing

```
>prove -vl t/module.t
```

- Adding *l* ensures it looks in the lib directory to find modules

Module Testing

```
t/module.t ..
1..3
ok 1 - use DnaHelpers;
ok 2 - Methionine found
ok 3 - Methionine not found
ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.05 usr  0.02 sys +
0.04 cusr  0.02 csys =  0.13 CPU)
Result: PASS
```

Object Testing

We can test an object.

- It's very similar to testing a module, except that we will be testing calls on an object.
- Lets test our PerlSequencer can actually produce reads of a given length

t/object.t

Object Testing

```
use Test::More tests => 203;
use_ok( 'PerlSequencer' );
```

- As we just *use* an object, we use the same test to ensure it compiles

Object Testing

```
my $sequencer = PerlSequencer->new();
```

- We will need to create an object to test it's modules

```
isa_ok( $sequencer, 'PerlSequencer', 'object' );
```

- isa_ok will check the object is of the given type

Object Testing

```
my @sequences =
 $sequencer->produce_reads( 25 );
```

- capture some data from the method we want to run

```
is( scalar @sequences, 200,
 q{produce reads produced 200 reads} );
```

- we expect that it will always produce 200 reads, so check the number of sequences obtained

Object Testing

```
my $count = 1;
foreach my $sequence ( @sequences ) {
  is( length $sequence, 25,
      "sequence $count has length 25" );
  $count++;
}
```

- loop through all the array, checking that every sequence has been produced with the expected read length

Object Testing

```
>prove -vl t/object.t
t/object.t ..
1..203
ok 1 - use PerlSequencer;
ok 2 - object isa PerlSequencer
ok 3 - produce reads produced 200 reads
ok 4 - sequence 1 has length 25 ...
ok 203 - sequence 200 has length 25
ok
All tests successful.
Files=1, Tests=203,  2 wallclock secs ( )
Result: PASS
```

Tests that fail

- Of course, so far everything has passed.
- What do we see when it fails?
- prove and Test::More combine to inform us.

t/fail.t

```
use Test::More tests => 1;
is( 'true', 'false', 'We expect this to fail' );
```

- We know that true and false are different, so we have made this test fail. Lets see it

Tests that fail

```
>prove -vl t/fail.t
t/fail.t ..
1..1
not ok 1 - We expect this to fail
#   Failed test 'We expect this to fail'
#   at t/fail.t line 3.
#          got: 'true'
#     expected: 'false'
# Looks like you failed 1 test of 1.
```

Tests that fail

```
Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/1 subtests
Test Summary Report
-------------------
t/fail.t (Wstat: 256 Tests: 1 Failed: 1)
  Failed test:  1
  Non-zero exit status: 1
Files=1, Tests=1,  0 wallclock secs ( 0.03 usr  0.02 sys +  0.02
cusr  0.00 csys =  0.07 CPU)
Result: FAIL
```

Tests that fail

- Obviously, in this case, we know this was to fail, but if you know what part of the code you are testing, it can help you locate what went wrong.

It's Too Much Code, Why Do It?

- Having tests gives you a confidence that the work you have produced is
  - reliable - you have tested what happens if different parameters are given
  - maintainable - if you want to modify the code (add more features, refactor) then if your tests still pass you know that you haven't broken it

It's Too Much Code, Why Do It?

  - User confidence – your tests pass, so your users can have confidence to use the code
  - Somewhere to find what it happening – others and you can look at the tests to find what should happen
  - Leads to Test Driven Development – which I could go on for hours about

Testing - Summary

- Tests automate checking code works
- They give you confidence
- They give users confidence

The End

And now, the end is near..

- I hope that you have found the course useful
- Feedback – sc3 will mail round forms
  - Please fill them in honestly, this is the first time this course has been run
  - Be Nice :)

What next for you?

- As with spoken languages, you need to keep practising
- I hope that you can go away and write yourself, or your teams some perl scripts
  - Start small, work up
- Don't worry about all of the features shown in this course
- You can take a copy of all the course materials

I'm here

- I do have a job, but I'm here if you want to ask me for any help
  - ajb@sanger.ac.uk
- I also keep a blog of my software exploits
  - vampiresoftware.blogspot.com
- Slideshare presentations
  - www.slideshare.net/setitesuk
- Cpan
  - http://search.cpan.org/~setitesuk/

Thanks for Listening

- Thankyou for coming and staying the course
- I hope you have enjoyed it
- Go forth, and perl!