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..