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