A Perl Testing Tutorial

chromatic

Why to Test

ensure the code does what it should do

- no missing functionality
- no accidental functionality
- when your tests all pass, you're done

explore boundary conditions

- how does the code handle bad input?
- how does the code handle very bad input?

prevent (known) regressions

- no tested features can be broken accidentally
- no fixed bugs can reoccur

improve bug reports

- test output can pinpoint errors
- easily generated by end users

```
run this and send me the output
```

enable refactoring

- you break it, you know it right away
- side effects immediately evident
- well-tested code tends to be more modular
  - I had several pre-existing modules with similar behavior to test. They were only tangentially related. After the second test suite turned out almost the same, I realized I needed a parent class. I moved most of the tests to the parent class and the inherited classes ended up much simpler and very much shorter. Their tests were much simpler, too. This gave me less code to maintain, fewer tests to write, and more flexibility for adding new features.

improve API

- hard to test code may be hard to use
- writing a test first forces you to use the interface before you create it
  - simpler interfaces are easier to test
  - smaller functions are easier to reuse

promote automation

- easy to test code is usually well-decoupled
- easy to test code is, by definition, scriptable

ease maintainability

- tests are a safety net
  - if you don't understand this by now, I'll repeat it once again
- well-written tests serve as a sort of documentation
  - demonstrate what features the software should have
  - demonstrate what behavior the code should express

Don't be Scared!

Don't be Scared!

- writing good tests exercises the same muscles as writing good code
- if you can write code, you can write tests
- if you can write good code, you can be taught to write good tests

Testing Has Psychological Benefits

- writing tests (especially test first) makes it easier to write tests
- well-tested code is more fun to hack
- test-first programming gives **instant** feedback
- that's instant **positive** feedback, most of the time

Perl makes it easy

- ExtUtils::MakeMaker and the 'test' target
- Test::* modules hide the Test::Harness expectations
- if lazy dopes like us can do it, you can too!

Basic Tests

Thinking "ok"

```
print "ok";
print "not ok";
```

- basis for all tests
- a simple boolean condition
- generally numbered sequentially

More thinking "ok"

- why printing is a loss:
  - same arguments as using subroutines (duplicate code)
  - manual test renumbering
  - no diagnostics unless you provide them
    - what matched
    - what didn't match
    - which test failed and where
  - no built-in provision for more complex constructs
  - can be hard to maintain

The ok() function

```
ok( 1 == 1 );
```

- basis for all Test::Simple, Test::More, Test::Builder tests
- a simple boolean condition
- why it's a win:
  - alleviates duplicate code
  - associates a name with a test
  - expresses test intent more clearly
  - avoids portability concerns

Test::Simple and its API

planning tests

- helps catch too many/not enough test errors
- helps identify which test fails
- required by Test::Harness

```
use Test::Simple tests => 10;
use Test::Simple 'no_plan';
```

tests => *n*

- produces 1..*n* output for Test::Harness
- should be set when the test suite is complete

no_plan

- produces 1..*actual* output at the end of the test
- when you don't know how many tests will run
- very handy while developing a module
- not so handy when distributing a module

ok()

- scalar context of operands
- first argument always true or false
- second argument optional, test name
- condition is code expression of the test name
- test name is a declaration of the intent of the condition

ok() passing

```
ok( 1, '1 should be true' );
```

```
ok 1 - 1 should be true
```

ok() failing

```
ok( 0, '0 should be true' );

not ok 2 - 0 should be true
#     Failed test (demotest.pl at line 17)
```

success is not a gray area (individual tests)

- pass, or fail, there is no maybe
- test the smallest piece possible
- test one thing at a time
- pass quietly
  - avoid errors
  - avoid warnings
  - test for errors and warnings if you must

success is all or nothing (test suite)

- stamp out expected failures
  - undermine confidence in the code
  - clutter up screens and logs
  - hide new problems
- keeping changes small helps localize damage
- keeping tests working helps localize damage
- there are ways to isolate platform-specific tests

Integrating testing with the development process

catching regressions

- most bugs aren't new, they happen again and again
- why let them reoccur?
- write a test for every new bug

testing first

- teeny, tiny, mini-iterations
- break each task into boolean expressions
- "What feature do I need next?"
  - smallest element of overall task
  - one small step at a time

The two test-first questions

- "How can I prove that this feature works?"
  - the simplest test that will fail unless the feature works
  - THE TEST MUST FAIL
- "What is the least amount of code I can write to pass the test?"
  - the simpler the test, the simpler the code you need
  - THE TEST MUST NOW PASS
- this produces known good code and a comprehensive test suite
- be sure to run the entire test suite after you implement a task

writing testable code

- the smaller the unit, the better
- the simpler the unit, the better
- the better the test suite, the easier it is to refactor
- writing your tests first promotes this!

distributing tests

- Test.pm is easier to distribute
  - single module, does not depend on Test::Builder
  - compatible with Test::More API
- Test::Simple suite can be bundled with module
  - CGI.pm does this, among others
- Test::Simple ships with 5.8
- a module can depend on Test::Simple

Exploring the Test::More API

graduating from Test::Simple to Test::More, or why ok() is not enough

- one bit of data -- either it passed or it failed
- contains little debugging information
  - what did you get?
  - what did you receive?
  - do you really want to exchange "add a print at line 43" emails?
- true laziness is a sign of good testing

matching things - is()

- asks the question "are these two things equal?"
- does a pretty good job of guessing at the **type** of equality
- takes an optional test name, so use one
- special cased for 'undef'

is() passing

```
is( 2 + 2, 4, 'Two plus two should be four' );

ok 1 - Two plus two should be four
```

is() failing

```
is( 2 + 2, 5, 'Two plus two should be five' );

not ok 2 - Two plus two should be five
#     Failed test (demotest.pl at line 8)
#          got: '4'
#     expected: '5'
```

matching things - isnt()

- asks the question "are these two things not equal?"
- essentially the opposite of is()

isnt() passing

```
isnt( 2 + 2, 5, 'Two plus two should not be five' );
```

```
ok 3 - Two plus two should not be five
```

isnt() failing

```
isnt( 2 + 2, 4, 'Two plus two should not be four' );

not ok 4 - Two plus two should not be four
#     Failed test (demotest.pl at line 10)
#     '4'
#         ne
#     '4'
1..4
```

matching things - like()

- applies a regular expression to a string
- first argument is the string
- second argument is a regular expression
  - compiled with qr//, where Perl version supports that
  - a string that looks like a regex '/foo/', otherwise

like() passing

```
like( 'foobar', qr/fo+/,
 '+ quantifier should match one or more' );

ok 5 - + quantifier should match one or more
```

like() failing

```
like( 'fbar', qr/fo+/,
 '+ quantifier should match zero or more' );

not ok 6 - + quantifier should match zero or more
#     Failed test (demotest.pl at line 12)
#                   'fbar'
#     doesn't match '(?-xism:fo+)'
```

matching things - unlike()

- essentially the opposite of like()
- the same rules apply

unlike() passing

```
unlike( 'foobar', qr/baz/, 'foobar should not match baz' );
```

```
ok 7 - foobar should not match baz
```

unlike() failing

```
unlike( 'foobar', qr/bar/, 'foobar should not match bar' );

not ok 8 - foobar should not match bar
#     Failed test (demotest.pl at line 14)
#                   'foobar'
#           matches '(?-xism:bar)'
1..8
```

exploring behavior - use_ok()

- (these tests all provide their own test names)
- see if a module compiles and can be import()ed
- load a module and import() it
- activates at runtime, not compile time!
- provides a default test name so you don't have to
- allows optional imports to be passed

use_ok() passing

```
use_ok( 'Test::More' );

ok 11 - use Test::More;
```

use_ok() failing

```
use_ok( 'No::Module' );

not ok 12 - use No::Module;
#     Failed test (demotest.pl at line 20)
#     Tried to use 'No::Module'.
#     Error:  Can't locate No/Module.pm in @INC (@INC
#     contains: /usr/lib/perl5/5.6.1/i386-linux
#     /usr/lib/perl5/5.6.1
#     /usr/lib/perl5/site_perl/5.6.1/i386-linux
#     /usr/lib/perl5/site_perl/5.6.1
#     /usr/lib/perl5/site_perl .) at (eval 6) line 2.
```

exploring behavior - require_ok()

- see if a module compiles
- load a module, but do not import() it
- also provides a default test name

require_ok() passing

```
require_ok( 'Test::Simple' );
```

```
ok 13 - require Test::Simple;
```

require_ok() failing

```
require_ok( 'No::Module' );

not ok 14 - require No::Module;
#     Failed test (demotest.pl at line 23)
#     Tried to require 'No::Module'.
#     Error:  Can't locate No/Module.pm in @INC (@INC
#     contains: /usr/lib/perl5/5.6.1/i386-linux
#     /usr/lib/perl5/5.6.1
#     /usr/lib/perl5/site_perl/5.6.1/i386-linux
#     /usr/lib/perl5/site_perl/5.6.1
#     /usr/lib/perl5/site_perl .) at (eval 8) line 2.
```

exploring behavior - isa_ok()

- see if an object is a type of reference
- can be an object
  - but not a class
  - it exists to save you from having to make three extra constructor tests
    - did it return something defined?
    - is that something a reference?
    - is that reference an object?
- respects inheritance
- can be a reference type (scalar, array, hash, io, code)

isa_ok() passing

```
isa_ok( [], 'ARRAY' );

ok 15 - The object isa ARRAY
```

isa_ok() failing

```
isa_ok( {}, 'IO::Socket' );

not ok 16 - The object isa IO::Socket
#     Failed test (demotest.pl at line 26)
#     The object isn't a 'IO::Socket' it's a 'HASH'
```

exploring behavior - can_ok()

- see if an object has named behavior(s)
- can be an object or a class
- pass several function or method names to test them all in a single test
- loop over a can_ok() call to test them all in multiple tests
- respects inheritance, but falls afoul of AUTOLOAD() just like UNIVERSAL::can() does

can_ok() passing

```
can_ok( 'Test::More', 'can_ok' );
ok 17 - Test::More->can('can_ok')
```

can_ok() failing

```
can_ok( 'Test::More', 'autotest' );

not ok 18 - Test::More->can('autotest')
#     Failed test (demotest.pl at line 29)
#     Test::More->can('autotest') failed
```

controlling things - skip()

- skips a specified number of tests
- generally used for things that will NEVER pass
- takes the reason to skip the tests and the number to skip
- should be in a labelled SKIP block
- should have a skip condition

skip() in action

```
SKIP: {
   skip( 'Never on a Sunday', 2 ) if (localtime)[6] == 0;
 can_ok( 'Perl::Porter', 'buy_beer' );
 can_ok( 'Perl::Porter', 'wear_short_pants' );
}

ok 21 # skip Never on a Sunday
ok 22 # skip Never on a Sunday
```

controlling things - todo()

- marks a specified number of tests as known failures
- generally used for failures
  - use sparingly!
  - use for features that aren't quite finished
  - use for bugs that aren't quite fixed
  - don't let these add up: they undermine the suite
- should be in a labelled TODO block
- must localize the $TODO variable with the name

todo() in action

```
TODO: {
 local $TODO = 'The alien overlords have not arrived';
 is( $smallpox->status(), 'ready', 'Smallpox ready' );
 ok( @cow_decoys > 1000, 'Bait ready...' );
}

not ok 23 - Smallpox ready # TODO The alien overlords have
not arrived
#     Failed (TODO) test (demotest.pl at line 46)
#          got: 'replenishing'
#     expected: 'ready'
not ok 24 - Bait ready... # TODO The alien overlords have
not arrived
#     Failed (TODO) test (demotest.pl at line 47)
```

controlling things - diag()

- display a (list of) diagnostic message(s)
- guaranteed not to interfere with the test harness
- gives no test number, pass, or fail
- useful for giving suggestions on tricky tests

diag() in action

```
diag( "Now that you know a bit about test writing,\n",
      "you have no excuse not to test." );

# Now that you know a bit about test writing,
# you have no excuse not to test.
```

controlling things - pass()

- make a test pass, unconditionally
- takes a test name -- that's it!

pass() in action

```
pass( 'Executive fiat should work' );

ok 19 - Executive fiat should work
```

controlling things - fail()

- makes a test fail, unconditionally
- also takes a test name, and that's it

fail() in action

```
fail( '... but we live in a meritocracy' );

not ok 20 - ... but we live in a meritocracy
#     Failed test (demotest.pl at line 32)
```