82% of Basic Perl Testing

Introduction

Perl has tremendous libraries for testing. Since 2001 we've seen Test::Simple, Test::More, Test::Harness, Test::Builder, Test::this_that_and_the_other thing.

Testing is clearly considered to be a best practice in the Perl community. For example, every distribution uploaded to the Comprehensive Perl Archive Network (CPAN) is expected to have a test suite.

Improving our testing capabilities is a strong part of Perl community culture. For example, at the end of March we had the fifth annual Perl QA Hackathon, held this year in Paris.

Many Perl developers, however, make little use of Perl's testing capabilities. If you are one of them, this presentation will give you the basics you need to begin writing tests for your code. Once you do the associated exercises, you will have 82% of what you need to be a competent tester in Perl.

To follow along, download this tarball now and extract it with the command indicated. You can also follow along with the handouts.

Test::Simple

A Perl test program is simply a Perl program that uses testing functions.

`Test::Simple::ok()`

We're going to start with an extremely simple program, `01_simple.t`, that uses the most basic of all Perl testing functions: the `ok()` function from core library Test::Simple.

We start by importing Test::Simple and telling it we plan to run just one test.

```
use Test::Simple ( tests => 1 );
```

We state a plan so that if the code dies while running the tests, we know how far we got. More on test plans later.

The `ok()` function takes two arguments.

```
$x = 1;
ok( $x, "'$x' evaluates to true in Perl" );
```

The first argument is mandatory: It's an expression which is to be evaluated for its truth in Perl terms. The second is not mandatory but is strongly recommended: It's a string which describes what the test is testing. You can refer to it as the

test description, the *test label* or the *test name*. What you call it doesn't matter as long as it's human-friendly.

The `ok()` function evaluates the expression as either *ok* or *not ok*. Everything else is semantic sugar.

Since `01_simple.t` is a Perl program, we can run it with Perl itself. The output looks like this:

```
1..1
ok 1 - '1' evaluates to true in Perl
```

The first line of output is the program's plan. The second line reports the result of test #1 -- which is *ok* -- and its description.

Testing for falsehood

With `02_simple.t` we add one more test: a test of an expression which should evaluate to false.

```
$x = 0;
ok( $x, "'$x' evaluates to false in Perl" );
```

Let's run it.

```
1..2
ok 1 - '1' evaluates to true in Perl
not ok 2 - '0' evaluates to false in Perl
#   Failed test ''0' evaluates to false in Perl'
#   at 02_simple.t line 11.
# Looks like you failed 1 test of 2.
```

The first line of output now reports that we ran two tests. But at the third line of output we have a *not ok* -- a test failure. This could be an error in the expression we're testing. Or it could be an error in the way we've written the test.

On closer inspection, we see that we have written the test incorrectly. The `ok()` function will report an *ok* only if its first argument evaluates to *true* in Perl terms. Since we assigned 0 to `$x`, `$x` evaluates to Perl *false*. Hence, we have to negate `$x` to make the expression Perl *true*. When we make this correction (`02_simple_corrected.t`) and re-run the test,

```
$x = 0;
# ok( $x, "'$x' evaluates to false in Perl" );

ok( ! $x, "'$x' evaluates to false in Perl" );
```

... we get the output we want.

```
1..2
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
```

Testing for numeric equality

In `03_simple.t` we have a more complex expression as the first argument to the next test.

```
$x = 4;
$y = 3;
$expected = 7;
ok( ( $x + $y == $expected ), "'$x' + '$y' adds up to '$expected'" );
```

We evaluate the whole expression, `$x + $y == $expected`. We're testing for numeric equality. 4 plus 3 equals 7; the expression is true; so we get an *ok*.

```
1..3
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
```

Note how the use of variables in our test description makes the program output more human-friendly.

Testing for string equality

In `04_simple.t`, we're testing for string equality rather than numeric equality. (Beginning with this slide, I'm omitting the tests we've already discussed from the slide. However, they're still in the test file we will run.)

```
$x = 'Love is a ';
$y = 'many splendored thing';
$expected = 'Love is a many splendored thing';
ok( ( $x .= $y ) eq $expected,
    "Concatenated to '$expected'" );
```

Note the `eq` operator instead of `==`. Note also that we have formulated the test as a comparison between something we *got* versus something which we *expected* to get. We'll see that the *got-expected* sequence of arguments is very typical in testing with Perl.

```
1..4
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
ok 4 - Concatenated to 'Love is a many splendored thing'
```

Testing a pattern match

In `05_simple.t` the expression we're evaluating for truth is a pattern match.

```
$string = 'Love is a many splendored thing';
ok( ($string =~ m/splendored/),
    "As expected, string matched pattern" );
```

Can the pattern `splendored` be found in the larger string? It can, so when we run the code we get an *ok*.

```
1..5
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
ok 4 - Concatenated to 'Love is a many splendored thing'
ok 5 - As expected, string matched pattern
```

What have we got so far? A Perl program which uses `Test::Simple:ok()` to test expressions for simple truth, numeric and string equality, and pattern matching.

Limitations of Test::Simple

Test::Simple does the job. I've written test suites for some of my CPAN libraries using nothing but Test::Simple and its `ok()` function.

But Test::Simple has two limitations. First, we have to count up the number of tests we've written and provide that number to Test::Simple:

```
use Test::Simple ( tests => 5 );
```

That's fine when we only have a few tests. But it gets annoying when we have lots of tests. Why can't we just let the program know when we're done testing and have the computer do the counting?

Second, the `ok()` function often requires us to cram a comparison between what we got and what we expected into a single expression which is then evaluated for its overall truth. That feels linguistically clumsy.

Test::More

We can do better by re-writing our tests using another Perl core library, Test::More. Test::More has the same `ok()` function as Test::Simple, but has a lot more besides.

Test plans and `done_testing()`

But we should first note that in Test::More we can avoid the need to count up the number of tests in our program by saying either:

```
use Test::More 'no_plan';
```

at the beginning of the program, or, better still, in modern versions of Test::More:

```
done_testing();
```

... at the end of the test program.

Testing for numeric equality with `Test::More::is()`

To see what Test::More has above and beyond Test::Simple, we'll rewrite the third of our five tests with Test::More. Let's look at `03_more.t`.

Our first two tests in `03_more.t` continue to use the `ok()` function, but we've rewritten the third test with `Test::More::is()`.

`Test::More::is()` takes three arguments: the first is the expression we're evaluating, the second is what we expected to get, and the third is the test description. `Test::More::is()` asks: Is what we got what we expected? `is()` figures out whether numeric or string equality is the appropriate basis for comparison.

```
$x = 4;                 $y = 3;             $expected = 7;
is(
    ($x + $y),                              # got
    $expected,                              # expected
    "'$x' + '$y' adds up to '$expected'"    # description
);
```

When we run `03_more.t`, we get the same output as we did when we ran `03_simple.t`:

```
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
1..3
```

... except that now our plan appears at the end of the output rather than the beginning.

Testing for string equality with `Test::More::is()`

Let's now turn to `04_more.t`, where we add a test for string equality:

```
$x = 'Love is a ';
$y = 'many splendored thing';
$expected = 'Love is a many splendored thing';
is(
    ( $x .= $y ),                   # got
    $expected,                      # expected
    "Concatenated to '$expected'"   # description
);
```

... we get output like that of `04_simple.t`:

```
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
ok 4 - Concatenated to 'Love is a many splendored thing'
1..4
```

Testing a pattern match with `Test::More::like()`

Finally, we can introduce a Test::More function specifically designed to test pattern matching: `Test::More::like()`:

```
$string = 'Love is a many splendored thing';
like(
    $string,                                # got
    qr/splendored/,                         # expected
    "As expected, string matched pattern"   # description
);
```

`Test::More::like()` has the same structure as `Test::More::is()`: three arguments. The first is a string that you get from running code. The second is a pattern -- a compiled regular expression -- that you expect the string to match against. The third is the test description. When we run the program, we get output similar to `05_simple.t`:

```
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
ok 4 - Concatenated to 'Love is a many splendored thing'
ok 5 - As expected, string matched pattern
1..5
```

We've now rewritten our tests using three Test::More functions instead of Test::Simple. Test::More has other functions besides `ok()`, `is()` and `like()`, but these are the most useful and we'll make more use of them in a moment.

Test harnesses

Let's pause, however, to look at a different way of running our tests. We said earlier that since Perl test programs are just Perl programs, they can be run with `perl` itself, as we have just done.

Usually, however, we want to run more than one test file at a time. To do so, it helps to use a program which is designed to serve as a *test harness* -- a program which figures out which tests need to be run, runs them, collects data about the tests and then reports the results.

`prove`

In Perl, the simplest test harness is the `prove` program created by Andy Lester and included in the Perl core library known as Test::Harness. Let's run both the Test::Simple and Test::More versions of our '05' tests.

```
$ prove t/05_simple.t t/05_more.t
t/05_simple.t .. ok
t/05_more.t .... ok
All tests successful.
Files=2, Tests=10,  0 wallclock secs
    ( 0.12 usr  0.04 sys +  0.15 cusr  0.05 csys =  0.36 CPU)
Result: PASS
```

Note that with `prove`, we get a summary reporting the number of test files run, the number of individual tests run, the time it took to run them and the overall result.

Like most programs used at the command-line, `prove` can take flags to modify its behavior. `prove` can also interpolate shell globs to figure out what files it needs to run. Let's run `prove` with the `-v` flag so that we get its *verbose* mode:

```
$ prove -v t/05*.t
t/05_simple.t ..
1..5
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
ok 4 - Concatenated to 'Love is a many splendored thing'
ok 5 - As expected, string matched pattern
ok
t/05_more.t ....
ok 1 - '1' evaluates to true in Perl
ok 2 - '0' evaluates to false in Perl
ok 3 - '4' + '3' adds up to '7'
ok 4 - Concatenated to 'Love is a many splendored thing'
```

```
ok 5 - As expected, string matched pattern
1..5
ok
All tests successful.
Files=2, Tests=10,  0 wallclock secs
    ( 0.13 usr  0.03 sys +  0.15 cusr  0.05 csys =  0.36 CPU)
Result: PASS
```

What have we learned so far? We've learned how to use Test::Simple and its `ok()` function; Test::More and its `ok()`, `is()`, `like()` and done_testing() functions; and the `prove` test harness program.

Testing Functions from a Perl Module

Let's now look at a more typical use of these functions: The testing of functions drawn from a Perl module. If we put subroutines into a `.pm` file, we can import them into any Perl program, including test programs. If we thoroughly test our subroutines in test programs, we will have more assurance that they are working properly in our production programs.

The Alpha module

In the tarball accompanying this presentation you will find a directory called `alpha`. The directory and files underneath `alpha` are set up like a typical Perl distribution on CPAN. Let's look first at the Perl module `lib/Alpha.pm`.

This is a very elementary module written in object-oriented Perl. The synopsis for `lib/Alpha.pm` shows that it has four methods which are called like this:

```
use Alpha;

$self = Alpha->new( {
    name    => 'Sylvester',     # optional
    string  => 'some string',
} );

$name = $self->get_name();

$string = $self->get_string();

$return_value = $self->is_valid_plang('Perl');
```

Alpha's constructor, `new()`, is documented as to its purpose, arguments and return value.

```
"new()"
```

```
        *   Purpose: Alpha constructor.

        *   Arguments: Reference to hash with two key-value pairs.

                $self = Alpha->new( {
                    name    => 'Sylvester', # optional; defaults to 'George'
                    string  => 'some string',
                } );

        *   Return Value: Alpha object.

        *   Comment: Internally defines a set of valid languages limited to:

                perl php prolog python

            ... spelled both that way and with initial upper-case letters.
```

`new()` takes arguments as key-value pairs inside a single hash reference.

```
sub new {
    my ($class, $args) = @_;

    $args->{name} ||= 'George';

    croak "Alpha::new() needs 'string' argument"
        unless (defined($args->{string}) and $args->{string} ne '');

    my @lc_languages = qw( perl php prolog python );
    my @languages = @lc_languages;
    for (@lc_languages) { push @languages, ucfirst($_); }
    $args->{languages} = { map { $_ => 1 } @languages };

    my $self = bless $args, $class;
    return $self;
}
```

The key `name` is optional -- it will default to `George`; the key `string`is not. The constructor will also store an element `languages` which treats `perl`, `php`, `prolog` and `python` as valid values whether spelled all-lower-case or initial-cap. We bless a reference to that hash into class and return the object.

We have 3 methods we can call on our object. `get_name()` simply reports either the `name` we passed to the constructor or the default value of `George`.

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

`get_string()` simply reports the `string` we passed to the constructor.

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

`is_valid_plang()` takes a string as an argument and reports whether that string is a valid *P-language*, *i.e.*, one of the 4 programming languages starting with *P* mentioned above.

```
sub is_valid_plang {
    my ($self, $lang) = @_;
    return unless defined $lang;
    return 0 unless $self->{languages}->{$lang};
    return 1;
}
```

Tests for the Alpha module

By convention, we store our tests in the `t/` subdirectory and name them with the `.t` extension. For our purposes today, I've combined them all the tests into a single test program, `t/all_methods.t`. Let's scroll through that file.

We first test `new()`, the constructor. Since the `name` element is optional, we test `new()` both with and without `name`.

```
    ##### new() #####

    # new(): typical case
    $self = Alpha->new( {
        name    => 'Sylvester',
        string  => 'some string',
    } );
    ok($self, "Alpha->new() returned true value");

    # new() without 'name' element
    $self = Alpha->new( {
        string  => 'some string',
    } );
    ok($self, "Alpha->new() returned true value even without 'name' argument");
```

The `ok()` function suffices for this.

We next test the `get_name()` method. We test for the case where we provided `name` to the constructor and for the case where we did not.

```
    ##### get_name() #####
```

```
$expected = 'Sylvester';

# new() with variable as value for 'name' element
$self = Alpha->new( {
    name    => $expected,
    string  => 'some string',
} );
ok($self, "Alpha->new() returned true value");

# get_name()
$got = $self->get_name();
is( $got, $expected, "get_name() returned '$expected' as expected" );

#new() with undefined value for 'name' element
$self = Alpha->new( {
    name    => undef,
    string  => 'some string',
} );
ok($self, "Alpha->new() returned true value");

# get_name() default case
$got = $self->get_name();
is( $got, 'George',
    "get_name() returned 'George' as expected in default case" );
```

Here we use `Test::More::is()` to test equality between strings.

We next test the `get_string()` method. We provide our constructor with an explicit value for `string` and test whether that value is returned by `get_string()`:

```
##### get_string() #####

$expected = 'abcxyz';

# new() without 'name' element
$self = Alpha->new( { string  => $expected } );
ok($self, "Alpha->new() returned true value");

# get_string()
$got = $self->get_string();
is( $got, $expected, "get_string() returned '$expected' as expected" );
```

We again use `Test::More::is()` to test equality between strings.

We next test the `is_valid_plang()` method.

```
##### is_valid_plang() #####

# new() without 'name' element
```

```
$self = Alpha->new( { string  => 'abcxyz' } );
ok($self, "Alpha->new() returned true value");

# is_valid_plang(): tests for several values of arguments
$language = 'perl';
ok($self->is_valid_plang($language), "'$language' is a valid language");

$language = 'Perl';
ok($self->is_valid_plang($language), "'$language' is a valid language");

$language = 'Python';
ok($self->is_valid_plang($language), "'$language' is a valid language");
```

There are actually eight different values for `$language` which will make the test pass, but these three cases will make the point. We should also make sure that if we fail to provide an argument to `is_valid_plang()`, the method returns an undefined value as per the documentation.

```
# is_valid_plang: test for no argument provided
ok(! defined($self->is_valid_plang()),
    "is_valid_plang() returned undefined");
```

We also need to test what happens when we provide `is_valid_plang()` with a value we do not expect to pass:

```
# is_valid_plang: test for invalid value (correctly written)
$language = 'abracadabra';
ok(! $self->is_valid_plang($language),
    "'$language' is not a valid language");
```

Note that we had to negate the return value of `is_valid_plang()` to get an expression which evaluates to Perl *true*. If you uncomment the following code, you will see what happens when we fail to do this:

```
# is_valid_plang: test for invalid value
# uncomment to see test FAIL
# $language = 'abracadabra';
# ok($self->is_valid_plang($language),
# "'$language' is a valid language");
```

Finally, we need to test certain conditions under which our code is so incorrect that the Alpha library calls for our program to `die`. According to the documentation, we must provide a *true* value to the `string` element in the hash reference passed to the constructor. We'll use `Test::More::like()` to see what happens when we provide *false* values to `new()`:

```
##### 'die' conditions #####

# new(): 'string' element undefined
# method will 'die'; capture error message
```

```
eval {
    $self = Alpha->new( {
        string  => undef,
    } );
};
like($@, qr/Alpha::new\(\) needs 'string' argument/,
    "Alpha->new(): got expected error message when 'string' was not defined");

# new(): 'string' element is empty string
# method will 'die'; capture error message
eval {
    $self = Alpha->new( {
        string  => '',
    } );
};
like($@, qr/Alpha::new\(\) needs 'string' argument/,
    "Alpha->new(): got expected error message when 'string' was empty");
```

Let's run our test file with `prove` in verbose mode:

```
$ prove -v t/all_methods.t
t/all_methods.t ..
ok 1 - Alpha->new() returned true value
ok 2 - Alpha->new() returned true value even without 'name' argument
ok 3 - Alpha->new() returned true value
ok 4 - get_name() returned 'Sylvester' as expected
ok 5 - Alpha->new() returned true value
ok 6 - get_name() returned 'George' as expected in default case
ok 7 - Alpha->new() returned true value
ok 8 - get_string() returned 'abcxyz' as expected
ok 9 - Alpha->new() returned true value
ok 10 - 'perl' is a valid language
ok 11 - 'Perl' is a valid language
ok 12 - 'Python' is a valid language
ok 13 - is_valid_plang() returned undefined
ok 14 - 'abracadabra' is not a valid language
ok 15 - Alpha->new(): got expected error message when 'string' was not defined
ok 16 - Alpha->new(): got expected error message when 'string' was empty
1..16
ok
All tests successful.
Files=1, Tests=16,  0 wallclock secs
    ( 0.12 usr  0.03 sys +  0.10 cusr  0.03 csys =  0.28 CPU)
Result: PASS
```

You now know Test::Simple, the Test::More functions `ok()`, `is()` and `like()`, the command-line utility `prove`. You know how to use them both directly on

code and on subroutines from Perl libraries. You now know at least 82% of what you need to be a competent Perl tester.

82% Competence in Perl Testing

Now at this point you are undoubtedly thinking: *"Where did you get that 82% figure from?"*

Since late 2006 I've spent a lot of time working on the Parrot virtual machine project. I'm not really fluent in the kind of heavily macroized C found in the Parrot guts -- or in the Perl 5 guts for that matter. So I mainly concentrate on writing tests for the parts of the Parrot distribution that are written in Perl 5. That includes the configuration system, many programs invoked by `make` during build, coding standards tests and more.

Over time I realized that just a few Test::More functions sufficed for the overwhelming majority of the tests I was writing or maintaining. I recently hacked up a version of Test::More which tallies the number of times any particular Test::More function is invoked over a given set of tests. These are the results:

```
ok                 1308
is                  703
like                281
isa_ok              267
pass                132
can_ok               55
is_deeply            29
isnt                 19
unlike                1
fail                  0
cmp_ok                0
use_ok                0
diag                  0
require_ok            0

Total              2795
ok/is/like %age    82.0
```

82% of what I need to accomplish during testing I can do using just three simple functions: `ok()`, `is()` and `like()`. I've noticed the same pattern in tests that I write on my day job. I am convinced that if you master Test::More's `ok()`, `is()` and `like()` functions and become experienced with using `prove`, you will be able to write tests for the overwhelming majority of the code you write as well.

Exercises/Bonus Slides

This concludes the formal part of this talk. At this point in the talk, we can either introduce you to some exercises or we can entertain questions about more advanced topics in Perl testing.

The 'Identifier' Library

In the same tarball in which you found this talk and the 'Alpha' library, you will find a subdirectory called `identifier/`.

```
cd identifier
perldoc lib/Identifier.pm
```

What you have in that directory is a Perl library with code, documentation in POD format, a `README` -- everything you need to put it up on CPAN except one important feature: tests. Its `t/` subdirectory contains only a placeholder file, `t/00.t`:

```
$ cat t/00.t
# placeholder for test files to be written by students
```

Your job is to read the documentation, study the code, write tests using Test::Simple and Test::More, and then run the tests with `prove`

```
$ prove -v t/*.t
```

If you want feedback on the tests you write, follow these steps:

```
perl Makefile.PL
make
make manifest
make test
make dist
```

... and then send the `.tgz` tarball file as an email attachment to me at:

```
jkeenan@cpan.org
```