NAME

Test - provides a simple framework for writing test scripts

SYNOPSIS

```
use strict;
use Test;

# use a BEGIN block so we print our plan before MyModule is loaded
BEGIN { plan tests => 14, todo => [3,4] }

# load your module...
use MyModule;

# Helpful notes.  All note-lines must start with a "#".
print "# I'm testing MyModule version $MyModule::VERSION\n";

ok(0); # failure
ok(1); # success

ok(0); # ok, expected failure (see todo list, above)
ok(1); # surprise success!

ok(0,1);             # failure: '0' ne '1'
ok('broke','fixed'); # failure: 'broke' ne 'fixed'
ok('fixed','fixed'); # success: 'fixed' eq 'fixed'
ok('fixed',qr/x/);   # success: 'fixed' =~ qr/x/

ok(sub { 1+1 }, 2);  # success: '2' eq '2'
ok(sub { 1+1 }, 3);  # failure: '2' ne '3'

my @list = (0,0);
ok @list, 3, "\@list=".join(',',@list);      #extra notes
ok 'segmentation fault', '/(?i)success/';    #regex match

skip(
  $^O =~ m/MSWin/ ? "Skip if MSWin" : 0,  # whether to skip
  $foo, $bar  # arguments just like for ok(...)
);
skip(
  $^O =~ m/MSWin/ ? 0 : "Skip unless MSWin",  # whether to skip
  $foo, $bar  # arguments just like for ok(...)
);
```

DESCRIPTION

This module simplifies the task of writing test files for Perl modules, such that their output is in the format that *Test::Harness* expects to see.

QUICK START GUIDE

To write a test for your new (and probably not even done) module, create a new file called *t/test.t* (in a new *t* directory). If you have multiple test files, to test the "foo", "bar", and "baz" feature sets, then feel free to call your files *t/foo.t*, *t/bar.t*, and *t/baz.t*

Functions

This module defines three public functions, `plan(...)`, `ok(...)`, and `skip(...)`. By default, all three are exported by the `use Test;` statement.

`plan(...)`

```
BEGIN { plan %theplan; }
```

This should be the first thing you call in your test script. It declares your testing plan, how many there will be, if any of them should be allowed to fail, and so on.

Typical usage is just:

```
use Test;
BEGIN { plan tests => 23 }
```

These are the things that you can put in the parameters to plan:

`tests =>` *`number`*

The number of tests in your script. This means all ok() and skip() calls.

`todo => [`*`1,5,14`*`]`

A reference to a list of tests which are allowed to fail. See *TODO TESTS*.

`onfail => sub { ... }`

`onfail => \&some_sub`

A subroutine reference to be run at the end of the test script, if any of the tests fail. See *ONFAIL*.

You must call `plan(...)` once and only once. You should call it in a `BEGIN {...}` block, like so:

```
BEGIN { plan tests => 23 }
```

_to_value

```
my $value = _to_value($input);
```

Converts an `ok` parameter to its value. Typically this just means running it, if it's a code reference. You should run all inputted values through this.

`ok(...)`

```
ok(1 + 1 == 2);
ok($have, $expect);
ok($have, $expect, $diagnostics);
```

This function is the reason for `Test`'s existence. It's the basic function that handles printing "`ok`" or "`not ok`", along with the current test number. (That's what `Test::Harness` wants to see.)

In its most basic usage, `ok(...)` simply takes a single scalar expression. If its value is true, the test passes; if false, the test fails. Examples:

```
# Examples of ok(scalar)

ok( 1 + 1 == 2 );           # ok if 1 + 1 == 2
ok( $foo =~ /bar/ );        # ok if $foo contains 'bar'
ok( baz($x + $y) eq 'Armondo' );    # ok if baz($x + $y) returns
                                    # 'Armondo'
ok( @a == @b );             # ok if @a and @b are the same length
```

The expression is evaluated in scalar context. So the following will work:

```
    ok( @stuff );                       # ok if @stuff has any
elements
    ok( !grep !defined $_, @stuff );    # ok if everything in @stuff
is
                                        # defined.
```

A special case is if the expression is a subroutine reference (in either `sub {...}` syntax or `\&foo` syntax). In that case, it is executed and its value (true or false) determines if the test passes or fails. For example,

```
    ok( sub {   # See whether sleep works at least passably
      my $start_time = time;
      sleep 5;
      time() - $start_time  >= 4
    });
```

In its two-argument form, `ok(`*`arg1`*`, `*`arg2`*`)` compares the two scalar values to see if they match. They match if both are undefined, or if *arg2* is a regex that matches *arg1*, or if they compare equal with `eq`.

```
    # Example of ok(scalar, scalar)

    ok( "this", "that" );               # not ok, 'this' ne 'that'
    ok( "", undef );                    # not ok, "" is defined
```

The second argument is considered a regex if it is either a regex object or a string that looks like a regex. Regex objects are constructed with the qr// operator in recent versions of perl. A string is considered to look like a regex if its first and last characters are "/", or if the first character is "m" and its second and last characters are both the same non-alphanumeric non-whitespace character. These regexp

Regex examples:

```
    ok( 'JaffO', '/Jaff/' );    # ok, 'JaffO' =~ /Jaff/
    ok( 'JaffO', 'm|Jaff|' );   # ok, 'JaffO' =~ m|Jaff|
    ok( 'JaffO', qr/Jaff/ );    # ok, 'JaffO' =~ qr/Jaff/;
    ok( 'JaffO', '/(?i)jaff/ ); # ok, 'JaffO' =~ /jaff/i;
```

If either (or both!) is a subroutine reference, it is run and used as the value for comparing. For example:

```
    ok sub {
        open(OUT, ">x.dat") || die $!;
        print OUT "\x{e000}";
        close OUT;
        my $bytecount = -s 'x.dat';
        unlink 'x.dat' or warn "Can't unlink : $!";
        return $bytecount;
      },
      4
    ;
```

The above test passes two values to `ok(arg1, arg2)` -- the first a coderef, and the second is the number 4. Before `ok` compares them, it calls the coderef, and uses its return value as the real value of this parameter. Assuming that `$bytecount` returns 4, `ok` ends up testing `4 eq 4`. Since that's true, this test passes.

Finally, you can append an optional third argument, in `ok(`*`arg1,arg2, note`*`)`, where *note* is a string value that will be printed if the test fails. This should be some useful information about

the test, pertaining to why it failed, and/or a description of the test. For example:

```
    ok( grep($_ eq 'something unique', @stuff), 1,
        "Something that should be unique isn't!\n".
        '@stuff = '.join ', ', @stuff
      );
```

Unfortunately, a note cannot be used with the single argument style of `ok()`. That is, if you try `ok(arg1, note)`, then `Test` will interpret this as `ok(arg1, arg2)`, and probably end up testing *arg1* `eq` *arg2* -- and that's not what you want!

All of the above special cases can occasionally cause some problems. See *BUGS and CAVEATS*.

`skip(skip_if_true, args...)`

This is used for tests that under some conditions can be skipped. It's basically equivalent to:

```
  if( $skip_if_true ) {
    ok(1);
  } else {
    ok( args... );
  }
```

...except that the `ok(1)` emits not just "`ok` *testnum*" but actually "`ok` *testnum* `#` *skip_if_true_value*".

The arguments after the *skip_if_true* are what is fed to `ok(...)` if this test isn't skipped.

Example usage:

```
  my $if_MSWin =
    $^O =~ m/MSWin/ ? 'Skip if under MSWin' : '';

  # A test to be skipped if under MSWin (i.e., run except under
MSWin)
  skip($if_MSWin, thing($foo), thing($bar) );
```

Or, going the other way:

```
  my $unless_MSWin =
    $^O =~ m/MSWin/ ? '' : 'Skip unless under MSWin';

  # A test to be skipped unless under MSWin (i.e., run only under
MSWin)
  skip($unless_MSWin, thing($foo), thing($bar) );
```

The tricky thing to remember is that the first parameter is true if you want to *skip* the test, not *run* it; and it also doubles as a note about why it's being skipped. So in the first codeblock above, read the code as "skip if MSWin -- (otherwise) test whether `thing($foo)` is `thing($bar)`" or for the second case, "skip unless MSWin...".

Also, when your *skip_if_reason* string is true, it really should (for backwards compatibility with older Test.pm versions) start with the string "Skip", as shown in the above examples.

Note that in the above cases, `thing($foo)` and `thing($bar)` *are* evaluated -- but as long as the `skip_if_true` is true, then we `skip(...)` just tosses out their value (i.e., not bothering to treat them like values to `ok(...)`. But if you need to *not* eval the arguments when skipping the test, use this format:

```
  skip( $unless_MSWin,
    sub {
      # This code returns true if the test passes.
      # (But it doesn't even get called if the test is skipped.)
```

```
    thing($foo) eq thing($bar)
  }
);
```

or even this, which is basically equivalent:

```
skip( $unless_MSWin,
  sub { thing($foo) }, sub { thing($bar) }
);
```

That is, both are like this:

```
if( $unless_MSWin ) {
  ok(1);  # but it actually appends "# $unless_MSWin"
          #  so that Test::Harness can tell it's a skip
} else {
  # Not skipping, so actually call and evaluate...
  ok( sub { thing($foo) }, sub { thing($bar) } );
}
```

TEST TYPES

* NORMAL TESTS

These tests are expected to succeed. Usually, most or all of your tests are in this category. If a normal test doesn't succeed, then that means that something is *wrong*.

* SKIPPED TESTS

The `skip(...)` function is for tests that might or might not be possible to run, depending on the availability of platform-specific features. The first argument should evaluate to true (think "yes, please skip") if the required feature is *not* available. After the first argument, `skip(...)` works exactly the same way as `ok(...)` does.

* TODO TESTS

TODO tests are designed for maintaining an **executable TODO list**. These tests are *expected to fail.* If a TODO test does succeed, then the feature in question shouldn't be on the TODO list, now should it?

Packages should NOT be released with succeeding TODO tests. As soon as a TODO test starts working, it should be promoted to a normal test, and the newly working feature should be documented in the release notes or in the change log.

ONFAIL

```
BEGIN { plan test => 4, onfail => sub { warn "CALL 911!" } }
```

Although test failures should be enough, extra diagnostics can be triggered at the end of a test run. `onfail` is passed an array ref of hash refs that describe each test failure. Each hash will contain at least the following fields: `package`, `repetition`, and `result`. (You shouldn't rely on any other fields being present.) If the test had an expected value or a diagnostic (or "note") string, these will also be included.

The *optional* `onfail` hook might be used simply to print out the version of your package and/or how to report problems. It might also be used to generate extremely sophisticated diagnostics for a particularly bizarre test failure. However it's not a panacea. Core dumps or other unrecoverable errors prevent the `onfail` hook from running. (It is run inside an `END` block.) Besides, `onfail` is probably over-kill in most cases. (Your test code should be simpler than the code it is testing, yes?)

BUGS and CAVEATS

- `ok(...)`'s special handing of strings which look like they might be regexes can also cause unexpected behavior. An innocent:

  ```
  ok( $fileglob, '/path/to/some/*stuff/' );
  ```

  will fail, since Test.pm considers the second argument to be a regex! The best bet is to use the one-argument form:

  ```
  ok( $fileglob eq '/path/to/some/*stuff/' );
  ```

- `ok(...)`'s use of string `eq` can sometimes cause odd problems when comparing numbers, especially if you're casting a string to a number:

  ```
  $foo = "1.0";
  ok( $foo, 1 );      # not ok, "1.0" ne 1
  ```

  Your best bet is to use the single argument form:

  ```
  ok( $foo == 1 );    # ok "1.0" == 1
  ```

- As you may have inferred from the above documentation and examples, `ok`'s prototype is `($;$$)` (and, incidentally, `skip`'s is `($;$$$)`). This means, for example, that you can do `ok @foo, @bar` to compare the *size* of the two arrays. But don't be fooled into thinking that `ok @foo, @bar` means a comparison of the contents of two arrays -- you're comparing *just* the number of elements of each. It's so easy to make that mistake in reading `ok @foo, @bar` that you might want to be very explicit about it, and instead write `ok scalar(@foo), scalar(@bar)`.

- This almost definitely doesn't do what you expect:

  ```
  ok $thingy->can('some_method');
  ```

  Why? Because `can` returns a coderef to mean "yes it can (and the method is this...)", and then `ok` sees a coderef and thinks you're passing a function that you want it to call and consider the truth of the result of! I.e., just like:

  ```
  ok $thingy->can('some_method')->();
  ```

  What you probably want instead is this:

  ```
  ok $thingy->can('some_method') && 1;
  ```

  If the `can` returns false, then that is passed to `ok`. If it returns true, then the larger expression `$thingy->can('some_method') && 1` returns 1, which `ok` sees as a simple signal of success, as you would expect.

- The syntax for `skip` is about the only way it can be, but it's still quite confusing. Just start with the above examples and you'll be okay.

  Moreover, users may expect this:

  ```
  skip $unless_mswin, foo($bar), baz($quux);
  ```

  to not evaluate `foo($bar)` and `baz($quux)` when the test is being skipped. But in reality, they *are* evaluated, but `skip` just won't bother comparing them if `$unless_mswin` is true.

  You could do this:

  ```
  skip $unless_mswin, sub{foo($bar)}, sub{baz($quux)};
  ```

  But that's not terribly pretty. You may find it simpler or clearer in the long run to just do things like this:

```
if( $^O =~ m/MSWin/ ) {
  print "# Yay, we're under $^O\n";
  ok foo($bar), baz($quux);
  ok thing($whatever), baz($stuff);
  ok blorp($quux, $whatever);
  ok foo($barzbarz), thang($quux);
} else {
  print "# Feh, we're under $^O.  Watch me skip some tests...\n";
  for(1 .. 4) { skip "Skip unless under MSWin" }
}
```

But be quite sure that `ok` is called exactly as many times in the first block as `skip` is called in the second block.

ENVIRONMENT

If `PERL_TEST_DIFF` environment variable is set, it will be used as a command for comparing unexpected multiline results. If you have GNU diff installed, you might want to set `PERL_TEST_DIFF` to `diff -u`. If you don't have a suitable program, you might install the `Text::Diff` module and then set `PERL_TEST_DIFF` to be `perl -MText::Diff -e 'print diff(@ARGV)'`. If `PERL_TEST_DIFF` isn't set but the `Algorithm::Diff` module is available, then it will be used to show the differences in multiline results.

NOTE

A past developer of this module once said that it was no longer being actively developed. However, rumors of its demise were greatly exaggerated. Feedback and suggestions are quite welcome.

Be aware that the main value of this module is its simplicity. Note that there are already more ambitious modules out there, such as *Test::More* and *Test::Unit*.

Some earlier versions of this module had docs with some confusing typos in the description of `skip(...)`.

SEE ALSO

Test::Harness

Test::Simple, *Test::More*, *Devel::Cover*

Test::Builder for building your own testing library.

Test::Unit is an interesting XUnit-style testing library.

Test::Inline and *SelfTest* let you embed tests in code.

AUTHOR

Current maintainer: Jesse Vincent. <jesse@bestpractical.com>