NAME

Test::More - yet another framework for writing test scripts

SYNOPSIS

```
  use Test::More tests => 23;
  # or
  use Test::More skip_all => $reason;
  # or
  use Test::More;   # see done_testing()

  require_ok( 'Some::Module' );

  # Various ways to say "ok"
  ok($got eq $expected, $test_name);

  is  ($got, $expected, $test_name);
  isnt($got, $expected, $test_name);

  # Rather than print STDERR "# here's what went wrong\n"
  diag("here's what went wrong");

  like  ($got, qr/expected/, $test_name);
  unlike($got, qr/expected/, $test_name);

  cmp_ok($got, '==', $expected, $test_name);

  is_deeply($got_complex_structure, $expected_complex_structure,
$test_name);

  SKIP: {
      skip $why, $how_many unless $have_some_feature;

      ok( foo(),       $test_name );
      is( foo(42), 23, $test_name );
  };

  TODO: {
      local $TODO = $why;

      ok( foo(),       $test_name );
      is( foo(42), 23, $test_name );
  };

  can_ok($module, @methods);
  isa_ok($object, $class);

  pass($test_name);
  fail($test_name);

  BAIL_OUT($why);
```

```
# UNIMPLEMENTED!!!
my @status = Test::More::status;
```

DESCRIPTION

STOP! If you're just getting started writing tests, have a look at *Test::Simple* first. This is a drop in replacement for Test::Simple which you can switch to once you get the hang of basic testing.

The purpose of this module is to provide a wide range of testing utilities. Various ways to say "ok" with better diagnostics, facilities to skip tests, test future features and compare complicated data structures. While you can do almost anything with a simple `ok()` function, it doesn't provide good diagnostic output.

I love it when a plan comes together

Before anything else, you need a testing plan. This basically declares how many tests your script is going to run to protect against premature failure.

The preferred way to do this is to declare a plan when you `use Test::More`.

```
use Test::More tests => 23;
```

There are cases when you will not know beforehand how many tests your script is going to run. In this case, you can declare your tests at the end.

```
use Test::More;

... run your tests ...

done_testing( $number_of_tests_run );
```

Sometimes you really don't know how many tests were run, or it's too difficult to calculate. In which case you can leave off $number_of_tests_run.

In some cases, you'll want to completely skip an entire testing script.

```
use Test::More skip_all => $skip_reason;
```

Your script will declare a skip with the reason why you skipped and exit immediately with a zero (success). See *Test::Harness* for details.

If you want to control what functions Test::More will export, you have to use the 'import' option. For example, to import everything but 'fail', you'd do:

```
use Test::More tests => 23, import => ['!fail'];
```

Alternatively, you can use the `plan()` function. Useful for when you have to calculate the number of tests.

```
use Test::More;
plan tests => keys %Stuff * 3;
```

or for deciding between running the tests at all:

```
use Test::More;
if( $^O eq 'MacOS' ) {
    plan skip_all => 'Test irrelevant on MacOS';
}
```

```
else {
    plan tests => 42;
}
```

done_testing

```
done_testing();
done_testing($number_of_tests);
```

If you don't know how many tests you're going to run, you can issue the plan when you're done running tests.

$number_of_tests is the same as `plan()`, it's the number of tests you expected to run. You can omit this, in which case the number of tests you ran doesn't matter, just the fact that your tests ran to conclusion.

This is safer than and replaces the "no_plan" plan.

Test names

By convention, each test is assigned a number in order. This is largely done automatically for you. However, it's often very useful to assign a name to each test. Which would you rather see:

```
ok 4
not ok 5
ok 6
```

or

```
ok 4 - basic multi-variable
not ok 5 - simple exponential
ok 6 - force == mass * acceleration
```

The later gives you some idea of what failed. It also makes it easier to find the test in your script, simply search for "simple exponential".

All test functions take a name argument. It's optional, but highly suggested that you use it.

I'm ok, you're not ok.

The basic purpose of this module is to print out either "ok #" or "not ok #" depending on if a given test succeeded or failed. Everything else is just gravy.

All of the following print "ok" or "not ok" depending on if the test succeeded or failed. They all also return true or false, respectively.

ok

```
ok($got eq $expected, $test_name);
```

This simply evaluates any expression (`$got eq $expected` is just a simple example) and uses that to determine if the test succeeded or failed. A true expression passes, a false one fails. Very simple.

For example:

```
ok( $exp{9} == 81,                   'simple exponential' );
ok( Film->can('db_Main'),            'set_db()' );
ok( $p->tests == 4,                  'saw tests' );
ok( !grep(!defined $_, @items),      'all items defined' );
```

(Mnemonic: "This is ok.")

$test_name is a very short description of the test that will be printed out. It makes it very easy to find a test in your script when it fails and gives others an idea of your intentions. $test_name is optional, but we **very** strongly encourage its use.

Should an `ok()` fail, it will produce some diagnostics:

```
not ok 18 - sufficient mucus
#   Failed test 'sufficient mucus'
#   in foo.t at line 42.
```

This is the same as *Test::Simple*'s `ok()` routine.

is

isnt

```
is  ( $got, $expected, $test_name );
isnt( $got, $expected, $test_name );
```

Similar to `ok()`, `is()` and `isnt()` compare their two arguments with `eq` and `ne` respectively and use the result of that to determine if the test succeeded or failed. So these:

```
# Is the ultimate answer 42?
is( ultimate_answer(), 42,          "Meaning of Life" );

# $foo isn't empty
isnt( $foo, '',     "Got some foo" );
```

are similar to these:

```
ok( ultimate_answer() eq 42,        "Meaning of Life" );
ok( $foo ne '',     "Got some foo" );
```

`undef` will only ever match `undef`. So you can test a value against `undef` like this:

```
is($not_defined, undef, "undefined as expected");
```

(Mnemonic: "This is that." "This isn't that.")

So why use these? They produce better diagnostics on failure. `ok()` cannot know what you are testing for (beyond the name), but `is()` and `isnt()` know what the test was and why it failed. For example this test:

```
my $foo = 'waffle';  my $bar = 'yarblokos';
is( $foo, $bar,   'Is foo the same as bar?' );
```

Will produce something like this:

```
not ok 17 - Is foo the same as bar?
#   Failed test 'Is foo the same as bar?'
#   in foo.t at line 139.
#          got: 'waffle'
#     expected: 'yarblokos'
```

So you can figure out what went wrong without rerunning the test.

You are encouraged to use `is()` and `isnt()` over `ok()` where possible, however do not be tempted to use them to find out if something is true or false!

```
# XXX BAD!
is( exists $brooklyn{tree}, 1, 'A tree grows in Brooklyn' );
```

This does not check if `exists $brooklyn{tree}` is true, it checks if it returns 1. Very different. Similar caveats exist for false and 0. In these cases, use `ok()`.

```
  ok( exists $brooklyn{tree},    'A tree grows in Brooklyn' );
```

A simple call to `isnt()` usually does not provide a strong test but there are cases when you cannot say much more about a value than that it is different from some other value:

```
  new_ok $obj, "Foo";

  my $clone = $obj->clone;
  isa_ok $obj, "Foo", "Foo->clone";

  isnt $obj, $clone, "clone() produces a different object";
```

For those grammatical pedants out there, there's an `isn't()` function which is an alias of `isnt()`.

like

```
  like( $got, qr/expected/, $test_name );
```

Similar to `ok()`, `like()` matches $got against the regex `qr/expected/`.

So this:

```
    like($got, qr/expected/, 'this is like that');
```

is similar to:

```
    ok( $got =~ m/expected/, 'this is like that');
```

(Mnemonic "This is like that".)

The second argument is a regular expression. It may be given as a regex reference (i.e. `qr//`) or (for better compatibility with older perls) as a string that looks like a regex (alternative delimiters are currently not supported):

```
    like( $got, '/expected/', 'this is like that' );
```

Regex options may be placed on the end (`'/expected/i'`).

Its advantages over `ok()` are similar to that of `is()` and `isnt()`. Better diagnostics on failure.

unlike

```
  unlike( $got, qr/expected/, $test_name );
```

Works exactly as `like()`, only it checks if $got **does not** match the given pattern.

cmp_ok

```
  cmp_ok( $got, $op, $expected, $test_name );
```

Halfway between `ok()` and `is()` lies `cmp_ok()`. This allows you to compare two arguments using any binary perl operator. The test passes if the comparison is true and fails otherwise.

```
    # ok( $got eq $expected );
    cmp_ok( $got, 'eq', $expected, 'this eq that' );

    # ok( $got == $expected );
    cmp_ok( $got, '==', $expected, 'this == that' );

    # ok( $got && $expected );
    cmp_ok( $got, '&&', $expected, 'this && that' );
    ...etc...
```

Its advantage over `ok()` is when the test fails you'll know what $got and $expected were:

```
not ok 1
#   Failed test in foo.t at line 12.
#     '23'
#         &&
#     undef
```

It's also useful in those cases where you are comparing numbers and `is()`'s use of `eq` will interfere:

```
cmp_ok( $big_hairy_number, '==', $another_big_hairy_number );
```

It's especially useful when comparing greater-than or smaller-than relation between values:

```
cmp_ok( $some_value, '<=', $upper_limit );
```

can_ok

```
can_ok($module, @methods);
can_ok($object, @methods);
```

Checks to make sure the $module or $object can do these @methods (works with functions, too).

```
can_ok('Foo', qw(this that whatever));
```

is almost exactly like saying:

```
ok( Foo->can('this') &&
    Foo->can('that') &&
    Foo->can('whatever')
  );
```

only without all the typing and with a better interface. Handy for quickly testing an interface.

No matter how many @methods you check, a single `can_ok()` call counts as one test. If you desire otherwise, use:

```
foreach my $meth (@methods) {
    can_ok('Foo', $meth);
}
```

isa_ok

```
isa_ok($object,   $class, $object_name);
isa_ok($subclass, $class, $object_name);
isa_ok($ref,      $type,  $ref_name);
```

Checks to see if the given `$object->isa($class)`. Also checks to make sure the object was defined in the first place. Handy for this sort of thing:

```
my $obj = Some::Module->new;
isa_ok( $obj, 'Some::Module' );
```

where you'd otherwise have to write

```
my $obj = Some::Module->new;
ok( defined $obj && $obj->isa('Some::Module') );
```

to safeguard against your test script blowing up.

You can also test a class, to make sure that it has the right ancestor:

```
    isa_ok( 'Vole', 'Rodent' );
```

It works on references, too:

```
    isa_ok( $array_ref, 'ARRAY' );
```

The diagnostics of this test normally just refer to 'the object'. If you'd like them to be more specific, you can supply an $object_name (for example 'Test customer').

new_ok

```
  my $obj = new_ok( $class );
  my $obj = new_ok( $class => \@args );
  my $obj = new_ok( $class => \@args, $object_name );
```

A convenience function which combines creating an object and calling `isa_ok()` on that object.

It is basically equivalent to:

```
    my $obj = $class->new(@args);
    isa_ok $obj, $class, $object_name;
```

If @args is not given, an empty list will be used.

This function only works on `new()` and it assumes `new()` will return just a single object which isa `$class`.

subtest

```
    subtest $name => \&code;
```

`subtest()` runs the &code as its own little test with its own plan and its own result. The main test counts this as a single test using the result of the whole subtest to determine if its ok or not ok.

For example...

```
  use Test::More tests => 3;

  pass("First test");

  subtest 'An example subtest' => sub {
      plan tests => 2;

      pass("This is a subtest");
      pass("So is this");
  };

  pass("Third test");
```

This would produce.

```
  1..3
  ok 1 - First test
      # Subtest: An example subtest
      1..2
      ok 1 - This is a subtest
      ok 2 - So is this
  ok 2 - An example subtest
  ok 3 - Third test
```

A subtest may call `skip_all`. No tests will be run, but the subtest is considered a skip.

```
    subtest 'skippy' => sub {
        plan skip_all => 'cuz I said so';
        pass('this test will never be run');
    };
```

Returns true if the subtest passed, false otherwise.

Due to how subtests work, you may omit a plan if you desire. This adds an implicit `done_testing()` to the end of your subtest. The following two subtests are equivalent:

```
    subtest 'subtest with implicit done_testing()', sub {
        ok 1, 'subtests with an implicit done testing should work';
        ok 1, '... and support more than one test';
        ok 1, '... no matter how many tests are run';
    };

    subtest 'subtest with explicit done_testing()', sub {
        ok 1, 'subtests with an explicit done testing should work';
        ok 1, '... and support more than one test';
        ok 1, '... no matter how many tests are run';
        done_testing();
    };
```

pass

fail

```
    pass($test_name);
    fail($test_name);
```

Sometimes you just want to say that the tests have passed. Usually the case is you've got some complicated condition that is difficult to wedge into an `ok()`. In this case, you can simply use `pass()` (to declare the test ok) or fail (for not ok). They are synonyms for `ok(1)` and `ok(0)`.

Use these very, very, very sparingly.

Module tests

Sometimes you want to test if a module, or a list of modules, can successfully load. For example, you'll often want a first test which simply loads all the modules in the distribution to make sure they work before going on to do more complicated testing.

For such purposes we have `use_ok` and `require_ok`.

require_ok

```
    require_ok($module);
    require_ok($file);
```

Tries to `require` the given $module or $file. If it loads successfully, the test will pass. Otherwise it fails and displays the load error.

`require_ok` will guess whether the input is a module name or a filename.

No exception will be thrown if the load fails.

```
    # require Some::Module
    require_ok "Some::Module";

    # require "Some/File.pl";
    require_ok "Some/File.pl";
```

```
# stop testing if any of your modules will not load
for my $module (@module) {
    require_ok $module or BAIL_OUT "Can't load $module";
}
```

use_ok

```
BEGIN { use_ok($module); }
BEGIN { use_ok($module, @imports); }
```

Like `require_ok`, but it will `use` the $module in question and only loads modules, not files.

If you just want to test a module can be loaded, use `require_ok`.

If you just want to load a module in a test, we recommend simply using `use` directly. It will cause the test to stop.

It's recommended that you run `use_ok()` inside a BEGIN block so its functions are exported at compile-time and prototypes are properly honored.

If @imports are given, they are passed through to the use. So this:

```
BEGIN { use_ok('Some::Module', qw(foo bar)) }
```

is like doing this:

```
use Some::Module qw(foo bar);
```

Version numbers can be checked like so:

```
# Just like "use Some::Module 1.02"
BEGIN { use_ok('Some::Module', 1.02) }
```

Don't try to do this:

```
BEGIN {
    use_ok('Some::Module');

    ...some code that depends on the use...
    ...happening at compile time...
}
```

because the notion of "compile-time" is relative. Instead, you want:

```
BEGIN { use_ok('Some::Module') }
BEGIN { ...some code that depends on the use... }
```

If you want the equivalent of `use Foo ()`, use a module but not import anything, use `require_ok`.

```
BEGIN { require_ok "Foo" }
```

Complex data structures

Not everything is a simple eq check or regex. There are times you need to see if two data structures are equivalent. For these instances Test::More provides a handful of useful functions.

NOTE I'm not quite sure what will happen with filehandles.

is_deeply

```
is_deeply( $got, $expected, $test_name );
```

Similar to `is()`, except that if $got and $expected are references, it does a deep comparison walking each data structure to see if they are equivalent. If the two structures are different, it

will display the place where they start differing.

`is_deeply()` compares the dereferenced values of references, the references themselves (except for their type) are ignored. This means aspects such as blessing and ties are not considered "different".

`is_deeply()` currently has very limited handling of function reference and globs. It merely checks if they have the same referent. This may improve in the future.

Test::Differences and *Test::Deep* provide more in-depth functionality along these lines.

Diagnostics

If you pick the right test function, you'll usually get a good idea of what went wrong when it failed. But sometimes it doesn't work out that way. So here we have ways for you to write your own diagnostic messages which are safer than just `print STDERR`.

diag

```
diag(@diagnostic_message);
```

Prints a diagnostic message which is guaranteed not to interfere with test output. Like `print` @diagnostic_message is simply concatenated together.

Returns false, so as to preserve failure.

Handy for this sort of thing:

```
ok( grep(/foo/, @users), "There's a foo user" ) or
    diag("Since there's no foo, check that /etc/bar is set up
right");
```

which would produce:

```
not ok 42 - There's a foo user
#   Failed test 'There's a foo user'
#   in foo.t at line 52.
# Since there's no foo, check that /etc/bar is set up right.
```

You might remember `ok() or diag()` with the mnemonic `open() or die()`.

NOTE The exact formatting of the diagnostic output is still changing, but it is guaranteed that whatever you throw at it won't interfere with the test.

note

```
note(@diagnostic_message);
```

Like `diag()`, except the message will not be seen when the test is run in a harness. It will only be visible in the verbose TAP stream.

Handy for putting in notes which might be useful for debugging, but don't indicate a problem.

```
note("Tempfile is $tempfile");
```

explain

```
my @dump = explain @diagnostic_message;
```

Will dump the contents of any references in a human readable format. Usually you want to pass this into `note` or `diag`.

Handy for things like...

```
is_deeply($have, $want) || diag explain $have;
```

or

```
    note explain \%args;
    Some::Class->method(%args);
```

Conditional tests

Sometimes running a test under certain conditions will cause the test script to die. A certain function or method isn't implemented (such as `fork()` on MacOS), some resource isn't available (like a net connection) or a module isn't available. In these cases it's necessary to skip tests, or declare that they are supposed to fail but will work in the future (a todo test).

For more details on the mechanics of skip and todo tests see *Test::Harness*.

The way Test::More handles this is with a named block. Basically, a block of tests which can be skipped over or made todo. It's best if I just show you...

SKIP: BLOCK

```
SKIP: {
    skip $why, $how_many if $condition;

    ...normal testing code goes here...
}
```

This declares a block of tests that might be skipped, $how_many tests there are, $why and under what $condition to skip them. An example is the easiest way to illustrate:

```
SKIP: {
    eval { require HTML::Lint };

    skip "HTML::Lint not installed", 2 if $@;

    my $lint = new HTML::Lint;
    isa_ok( $lint, "HTML::Lint" );

    $lint->parse( $html );
    is( $lint->errors, 0, "No errors found in HTML" );
}
```

If the user does not have HTML::Lint installed, the whole block of code *won't be run at all*. Test::More will output special ok's which Test::Harness interprets as skipped, but passing, tests.

It's important that $how_many accurately reflects the number of tests in the SKIP block so the # of tests run will match up with your plan. If your plan is `no_plan` $how_many is optional and will default to 1.

It's perfectly safe to nest SKIP blocks. Each SKIP block must have the label `SKIP`, or Test::More can't work its magic.

You don't skip tests which are failing because there's a bug in your program, or for which you don't yet have code written. For that you use TODO. Read on.

TODO: BLOCK

```
TODO: {
    local $TODO = $why if $condition;

    ...normal testing code goes here...
}
```

Declares a block of tests you expect to fail and $why. Perhaps it's because you haven't fixed a bug or haven't finished a new feature:

```
TODO: {
    local $TODO = "URI::Geller not finished";

    my $card = "Eight of clubs";
    is( URI::Geller->your_card, $card, 'Is THIS your card?' );

    my $spoon;
    URI::Geller->bend_spoon;
    is( $spoon, 'bent',    "Spoon bending, that's original" );
}
```

With a todo block, the tests inside are expected to fail. Test::More will run the tests normally, but print out special flags indicating they are "todo". *Test::Harness* will interpret failures as being ok. Should anything succeed, it will report it as an unexpected success. You then know the thing you had todo is done and can remove the TODO flag.

The nice part about todo tests, as opposed to simply commenting out a block of tests, is it's like having a programmatic todo list. You know how much work is left to be done, you're aware of what bugs there are, and you'll know immediately when they're fixed.

Once a todo test starts succeeding, simply move it outside the block. When the block is empty, delete it.

todo_skip

```
TODO: {
    todo_skip $why, $how_many if $condition;

    ...normal testing code...
}
```

With todo tests, it's best to have the tests actually run. That way you'll know when they start passing. Sometimes this isn't possible. Often a failing test will cause the whole program to die or hang, even inside an `eval BLOCK` with and using `alarm`. In these extreme cases you have no choice but to skip over the broken tests entirely.

The syntax and behavior is similar to a `SKIP: BLOCK` except the tests will be marked as failing but todo. *Test::Harness* will interpret them as passing.

When do I use SKIP vs. TODO?

If it's something the user might not be able to do, use SKIP. This includes optional modules that aren't installed, running under an OS that doesn't have some feature (like `fork()` or symlinks), or maybe you need an Internet connection and one isn't available.

If it's something the programmer hasn't done yet, use TODO. This is for any code you haven't written yet, or bugs you have yet to fix, but want to put tests in your testing script (always a good idea).

Test control

BAIL_OUT

```
BAIL_OUT($reason);
```

Indicates to the harness that things are going so badly all testing should terminate. This includes the running of any additional test scripts.

This is typically used when testing cannot continue such as a critical module failing to compile or a necessary external utility not being available such as a database connection failing.

The test will exit with 255.

For even better control look at *Test::Most*.

Discouraged comparison functions

The use of the following functions is discouraged as they are not actually testing functions and produce no diagnostics to help figure out what went wrong. They were written before `is_deeply()` existed because I couldn't figure out how to display a useful diff of two arbitrary data structures.

These functions are usually used inside an `ok()`.

```
ok( eq_array(\@got, \@expected) );
```

`is_deeply()` can do that better and with diagnostics.

```
is_deeply( \@got, \@expected );
```

They may be deprecated in future versions.

eq_array

```
my $is_eq = eq_array(\@got, \@expected);
```

Checks if two arrays are equivalent. This is a deep check, so multi-level structures are handled correctly.

eq_hash

```
my $is_eq = eq_hash(\%got, \%expected);
```

Determines if the two hashes contain the same keys and values. This is a deep check.

eq_set

```
my $is_eq = eq_set(\@got, \@expected);
```

Similar to `eq_array()`, except the order of the elements is **not** important. This is a deep check, but the irrelevancy of order only applies to the top level.

```
ok( eq_set(\@got, \@expected) );
```

Is better written:

```
is_deeply( [sort @got], [sort @expected] );
```

NOTE By historical accident, this is not a true set comparison. While the order of elements does not matter, duplicate elements do.

NOTE `eq_set()` does not know how to deal with references at the top level. The following is an example of a comparison which might not work:

```
eq_set([\1, \2], [\2, \1]);
```

Test::Deep contains much better set comparison functions.

Extending and Embedding Test::More

Sometimes the Test::More interface isn't quite enough. Fortunately, Test::More is built on top of *Test::Builder* which provides a single, unified backend for any test library to use. This means two test libraries which both use <Test::Builder> **can** be used together in the same program>.

If you simply want to do a little tweaking of how the tests behave, you can access the underlying *Test::Builder* object like so:

builder

```
my $test_builder = Test::More->builder;
```

Returns the *Test::Builder* object underlying Test::More for you to play with.

EXIT CODES

If all your tests passed, *Test::Builder* will exit with zero (which is normal). If anything failed it will exit with how many failed. If you run less (or more) tests than you planned, the missing (or extras) will be considered failures. If no tests were ever run *Test::Builder* will throw a warning and exit with 255. If the test died, even after having successfully completed all its tests, it will still be considered a failure and will exit with 255.

So the exit codes are...

```
0                   all tests successful
255                 test died or all passed but wrong # of tests run
any other number    how many failed (including missing or extras)
```

If you fail more than 254 tests, it will be reported as 254.

NOTE This behavior may go away in future versions.

COMPATIBILITY

Test::More works with Perls as old as 5.8.1.

Thread support is not very reliable before 5.10.1, but that's because threads are not very reliable before 5.10.1.

Although Test::More has been a core module in versions of Perl since 5.6.2, Test::More has evolved since then, and not all of the features you're used to will be present in the shipped version of Test::More. If you are writing a module, don't forget to indicate in your package metadata the minimum version of Test::More that you require. For instance, if you want to use `done_testing()` but want your test script to run on Perl 5.10.0, you will need to explicitly require Test::More > 0.88.

Key feature milestones include:

subtests

Subtests were released in Test::More 0.94, which came with Perl 5.12.0. Subtests did not implicitly call `done_testing()` until 0.96; the first Perl with that fix was Perl 5.14.0 with 0.98.

`done_testing()`

This was released in Test::More 0.88 and first shipped with Perl in 5.10.1 as part of Test::More 0.92.

`cmp_ok()`

Although `cmp_ok()` was introduced in 0.40, 0.86 fixed an important bug to make it safe for overloaded objects; the fixed first shipped with Perl in 5.10.1 as part of Test::More 0.92.

`new_ok()` `note()` and `explain()`

These were was released in Test::More 0.82, and first shipped with Perl in 5.10.1 as part of Test::More 0.92.

There is a full version history in the Changes file, and the Test::More versions included as core can be found using *Module::CoreList*:

```
$ corelist -a Test::More
```

CAVEATS and NOTES

utf8 / "Wide character in print"

If you use utf8 or other non-ASCII characters with Test::More you might get a "Wide character

in print" warning. Using `binmode STDOUT, ":utf8"` will not fix it. *Test::Builder* (which powers Test::More) duplicates STDOUT and STDERR. So any changes to them, including changing their output disciplines, will not be seem by Test::More.

One work around is to apply encodings to STDOUT and STDERR as early as possible and before Test::More (or any other Test module) loads.

```
use open ':std', ':encoding(utf8)';
use Test::More;
```

A more direct work around is to change the filehandles used by *Test::Builder*.

```
my $builder = Test::More->builder;
binmode $builder->output,         ":encoding(utf8)";
binmode $builder->failure_output, ":encoding(utf8)";
binmode $builder->todo_output,    ":encoding(utf8)";
```

Overloaded objects

String overloaded objects are compared **as strings** (or in `cmp_ok()`'s case, strings or numbers as appropriate to the comparison op). This prevents Test::More from piercing an object's interface allowing better blackbox testing. So if a function starts returning overloaded objects instead of bare strings your tests won't notice the difference. This is good.

However, it does mean that functions like `is_deeply()` cannot be used to test the internals of string overloaded objects. In this case I would suggest *Test::Deep* which contains more flexible testing functions for complex data structures.

Threads

Test::More will only be aware of threads if `use threads` has been done *before* Test::More is loaded. This is ok:

```
use threads;
use Test::More;
```

This may cause problems:

```
use Test::More
use threads;
```

5.8.1 and above are supported. Anything below that has too many bugs.

HISTORY

This is a case of convergent evolution with Joshua Pritikin's *Test* module. I was largely unaware of its existence when I'd first written my own `ok()` routines. This module exists because I can't figure out how to easily wedge test names into Test's interface (along with a few other problems).

The goal here is to have a testing utility that's simple to learn, quick to use and difficult to trip yourself up with while still providing more flexibility than the existing Test.pm. As such, the names of the most common routines are kept tiny, special cases and magic side-effects are kept to a minimum. WYSIWYG.

SEE ALSO

ALTERNATIVES

Test::Simple if all this confuses you and you just want to write some tests. You can upgrade to Test::More later (it's forward compatible).

Test::Legacy tests written with Test.pm, the original testing module, do not play well with other testing libraries. Test::Legacy emulates the Test.pm interface and does play well with others.

TESTING FRAMEWORKS

Fennec The Fennec framework is a testers toolbox. It uses *Test::Builder* under the hood. It brings enhancements for forking, defining state, and mocking. Fennec enhances several modules to work better together than they would if you loaded them individually on your own.

Fennec::Declare Provides enhanced (*Devel::Declare*) syntax for Fennec.

ADDITIONAL LIBRARIES

Test::Differences for more ways to test complex data structures. And it plays well with Test::More.

Test::Class is like xUnit but more perlish.

Test::Deep gives you more powerful complex data structure testing.

Test::Inline shows the idea of embedded testing.

Mock::Quick The ultimate mocking library. Easily spawn objects defined on the fly. Can also override, block, or reimplement packages as needed.

Test::FixtureBuilder Quickly define fixture data for unit tests.

OTHER COMPONENTS

Test::Harness is the test runner and output interpreter for Perl. It's the thing that powers `make test` and where the `prove` utility comes from.

BUNDLES

Bundle::Test installs a whole bunch of useful test modules.

Test::Most Most commonly needed test functions and features.

AUTHORS

Michael G Schwern <schwern@pobox.com> with much inspiration from Joshua Pritikin's Test module and lots of help from Barrie Slaymaker, Tony Bowden, blackstar.co.uk, chromatic, Fergal Daly and the perl-qa gang.

MAINTAINERS

Chad Granum <exodist@cpan.org>

BUGS

See *http://rt.cpan.org* to report and view bugs.

SOURCE

The source code repository for Test::More can be found at *http://github.com/Test-More/test-more/*.

COPYRIGHT