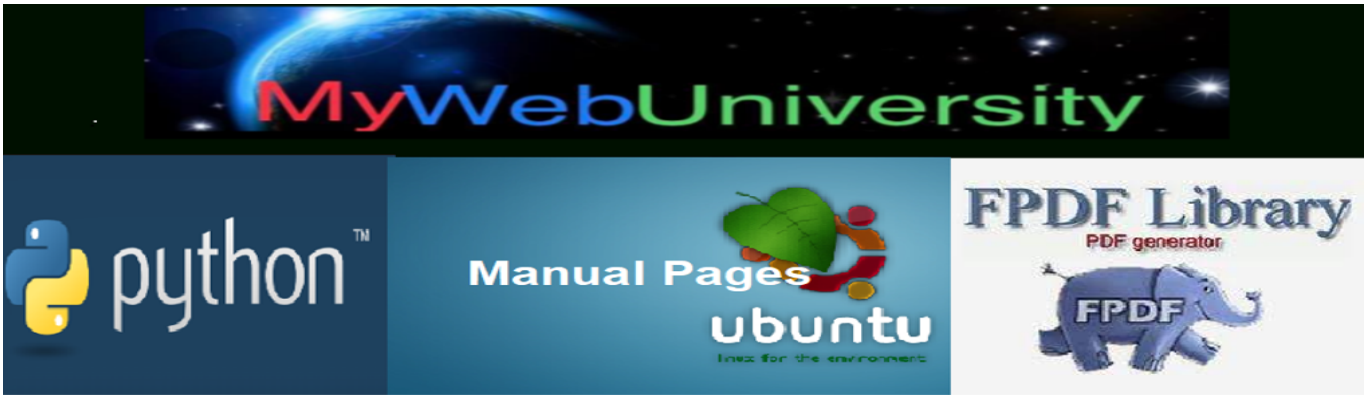

Rocky Enterprise Linux 9.2 Manual Pages on command 'Test::Tutorial.3perl'

$ man Test::Tutorial.3perl

Test::Tutorial(3perl) Perl Programmers Reference Guide Test::Tutorial(3perl)

NAME

Test::Tutorial - A tutorial about writing really basic tests

DESCRIPTION

AHHHHHHH!!!! NOT TESTING! Anything but testing! Beat me, whip me, send me to Detroit,

but don't make me write tests!

sob

Besides, I don't know how to write the damned things.

Is this you? Is writing tests right up there with writing documentation and having your

fingernails pulled out? Did you open up a test and read

######## We start with some black magic

and decide that's quite enough for you?

It's ok. That's all gone now. We've done all the black magic for you. And here are the

tricks...

Nuts and bolts of testing.

Here's the most basic test program.

#!/usr/bin/perl -w

print "1..1\n";

print 1 + 1 == 2 ? "ok 1\n" : "not ok 1\n";

Because 1 + 1 is 2, it prints:

1..1

ok 1

What this says is: 1..1 "I'm going to run one test." [1] "ok 1" "The first test passed".

And that's about all magic there is to testing. Your basic unit of testing is the ok.

For each thing you test, an "ok" is printed. Simple. Test::Harness interprets your test results to determine if you succeeded or failed (more on that later).

Writing all these print statements rapidly gets tedious. Fortunately, there's Test::Simple. It has one function, "ok()".

```
#!/usr/bin/perl -w

use Test::Simple tests => 1;

ok( 1 + 1 == 2 );
```

That does the same thing as the previous code. "ok()" is the backbone of Perl testing, and we'll be using it instead of roll-your-own from here on. If "ok()" gets a true value, the test passes. False, it fails.

```
#!/usr/bin/perl -w

use Test::Simple tests => 2;

ok( 1 + 1 == 2 );

ok( 2 + 2 == 5 );
```

From that comes:

```
1..2

ok 1

not ok 2

#     Failed test (test.pl at line 5)

# Looks like you failed 1 tests of 2.
```

1..2 "I'm going to run two tests." This number is a plan. It helps to ensure your test program ran all the way through and didn't die or skip some tests. "ok 1" "The first test passed." "not ok 2" "The second test failed". Test::Simple helpfully prints out some extra commentary about your tests.

It's not scary. Come, hold my hand. We're going to give an example of testing a module. For our example, we'll be testing a date library, Date::ICal. It's on CPAN, so download a copy and follow along. [2]

Where to start?

This is the hardest part of testing, where do you start? People often get overwhelmed at the apparent enormity of the task of testing a whole module. The best place to start is at the beginning. Date::ICal is an object-oriented module, and that means you start by making an object. Test "new()".

```
#!/usr/bin/perl -w
# assume these two lines are in all subsequent examples
use strict;
use warnings;
use Test::Simple tests => 2;
use Date::ICal;
my $ical = Date::ICal->new;         # create an object
ok( defined $ical );                # check that we got something
ok( $ical->isa('Date::ICal') );     # and it's the right class
```

Run that and you should get:

```
1..2
ok 1
ok 2
```

Congratulations! You've written your first useful test.

Names

That output isn't terribly descriptive, is it? When you have two tests you can figure out which one is #2, but what if you have 102 tests?

Each test can be given a little descriptive name as the second argument to "ok()".

```
use Test::Simple tests => 2;
ok( defined $ical,              'new() returned something' );
ok( $ical->isa('Date::ICal'),   "  and it's the right class" );
```

Now you'll see:

```
1..2
ok 1 - new() returned something
ok 2 -   and it's the right class
```

Test the manual

The simplest way to build up a decent testing suite is to just test what the manual says it does. [3] Let's pull something out of the "SYNOPSIS" in Date::ICal and test that all its bits work.

```
#!/usr/bin/perl -w
use Test::Simple tests => 8;
use Date::ICal;
$ical = Date::ICal->new( year => 1964, month => 10, day => 16,
```

```
                    hour => 16,   min   => 12, sec => 47,
                    tz   => '0530' );
    ok( defined $ical,            'new() returned something' );
    ok( $ical->isa('Date::ICal'), "  and it's the right class" );
    ok( $ical->sec   == 47,       '  sec()'   );
    ok( $ical->min   == 12,       '  min()'   );
    ok( $ical->hour  == 16,       '  hour()'  );
    ok( $ical->day   == 17,       '  day()'   );
    ok( $ical->month == 10,       '  month()' );
    ok( $ical->year  == 1964,     '  year()'  );
```

Run that and you get:

```
    1..8
    ok 1 - new() returned something
    ok 2 -   and it's the right class
    ok 3 -   sec()
    ok 4 -   min()
    ok 5 -   hour()
    not ok 6 -   day()
    #     Failed test (- at line 16)
    ok 7 -   month()
    ok 8 -   year()
    # Looks like you failed 1 tests of 8.
```

Whoops, a failure! [4] Test::Simple helpfully lets us know on what line the failure occurred, but not much else. We were supposed to get 17, but we didn't. What did we get?? Dunno. You could re-run the test in the debugger or throw in some print statements to find out.

Instead, switch from Test::Simple to Test::More. Test::More does everything Test::Simple does, and more! In fact, Test::More does things exactly the way Test::Simple does. You can literally swap Test::Simple out and put Test::More in its place. That's just what we're going to do.

Test::More does more than Test::Simple. The most important difference at this point is it provides more informative ways to say "ok". Although you can write almost any test with a generic "ok()", it can't tell you what went wrong. The "is()" function lets us declare

that something is supposed to be the same as something else:

```
use Test::More tests => 8;
use Date::ICal;
$ical = Date::ICal->new( year => 1964, month => 10, day => 16,
                         hour => 16,   min   => 12, sec => 47,
                         tz   => '0530' );
ok( defined $ical,            'new() returned something' );
ok( $ical->isa('Date::ICal'), "  and it's the right class" );
is( $ical->sec,     47,       '  sec()'   );
is( $ical->min,     12,       '  min()'   );
is( $ical->hour,    16,       '  hour()'  );
is( $ical->day,     17,       '  day()'   );
is( $ical->month,   10,       '  month()' );
is( $ical->year,    1964,     '  year()'  );
```

"Is "$ical->sec" 47?" "Is "$ical->min" 12?" With "is()" in place, you get more information:

```
1..8
ok 1 - new() returned something
ok 2 -   and it's the right class
ok 3 -   sec()
ok 4 -   min()
ok 5 -   hour()
not ok 6 -   day()
#     Failed test (- at line 16)
#          got: '16'
#     expected: '17'
ok 7 -   month()
ok 8 -   year()
# Looks like you failed 1 tests of 8.
```

Aha. "$ical->day" returned 16, but we expected 17. A quick check shows that the code is working fine, we made a mistake when writing the tests. Change it to:

```
is( $ical->day,     16,       '  day()'   );
```

... and everything works.

Any time you're doing a "this equals that" sort of test, use "is()". It even works on arrays. The test is always in scalar context, so you can test how many elements are in an array this way. [5]

```
is( @foo, 5, 'foo has 5 elements' );
```

Sometimes the tests are wrong

This brings up a very important lesson. Code has bugs. Tests are code. Ergo, tests have bugs. A failing test could mean a bug in the code, but don't discount the possibility that the test is wrong.

On the flip side, don't be tempted to prematurely declare a test incorrect just because you're having trouble finding the bug. Invalidating a test isn't something to be taken lightly, and don't use it as a cop out to avoid work.

Testing lots of values

We're going to be wanting to test a lot of dates here, trying to trick the code with lots of different edge cases. Does it work before 1970? After 2038? Before 1904? Do years after 10,000 give it trouble? Does it get leap years right? We could keep repeating the code above, or we could set up a little try/expect loop.

```
use Test::More tests => 32;
use Date::ICal;
my %ICal_Dates = (
        # An ICal string     And the year, month, day
        #                    hour, minute and second we expect.
        '19971024T120000' =>    # from the docs.
                        [ 1997, 10, 24, 12,  0,  0 ],
        '20390123T232832' =>    # after the Unix epoch
                        [ 2039,  1, 23, 23, 28, 32 ],
        '19671225T000000' =>    # before the Unix epoch
                        [ 1967, 12, 25,  0,  0,  0 ],
        '18990505T232323' =>    # before the MacOS epoch
                        [ 1899,  5,  5, 23, 23, 23 ],
);
while( my($ical_str, $expect) = each %ICal_Dates ) {
    my $ical = Date::ICal->new( ical => $ical_str );
    ok( defined $ical,            "new(ical => '$ical_str')" );
```

```
        ok( $ical->isa('Date::ICal'), "  and it's the right class" );
        is( $ical->year,    $expect->[0],     '  year()'  );
        is( $ical->month,   $expect->[1],     '  month()' );
        is( $ical->day,     $expect->[2],     '  day()'   );
        is( $ical->hour,    $expect->[3],     '  hour()'  );
        is( $ical->min,     $expect->[4],     '  min()'   );
        is( $ical->sec,     $expect->[5],     '  sec()'   );
    }
```

Now we can test bunches of dates by just adding them to %ICal_Dates. Now that it's less work to test with more dates, you'll be inclined to just throw more in as you think of them. Only problem is, every time we add to that we have to keep adjusting the "use Test::More tests => ##" line. That can rapidly get annoying. There are ways to make this work better.

First, we can calculate the plan dynamically using the "plan()" function.

```
    use Test::More;
    use Date::ICal;
    my %ICal_Dates = (
        ...same as before...
    );
    # For each key in the hash we're running 8 tests.
    plan tests => keys(%ICal_Dates) * 8;
    ...and then your tests...
```

To be even more flexible, use "done_testing". This means we're just running some tests, don't know how many. [6]

```
    use Test::More;   # instead of tests => 32
    ... # tests here
    done_testing();   # reached the end safely
```

If you don't specify a plan, Test::More expects to see "done_testing()" before your program exits. It will warn you if you forget it. You can give "done_testing()" an optional number of tests you expected to run, and if the number ran differs, Test::More will give you another kind of warning.

Informative names

Take a look at the line:

```
ok( defined $ical,            "new(ical => '$ical_str')" );
```

We've added more detail about what we're testing and the ICal string itself we're trying out to the name. So you get results like:

```
ok 25 - new(ical => '19971024T120000')
ok 26 -   and it's the right class
ok 27 -   year()
ok 28 -   month()
ok 29 -   day()
ok 30 -   hour()
ok 31 -   min()
ok 32 -   sec()
```

If something in there fails, you'll know which one it was and that will make tracking down the problem easier. Try to put a bit of debugging information into the test names. Describe what the tests test, to make debugging a failed test easier for you or for the next person who runs your test.

Skipping tests

Poking around in the existing Date::ICal tests, I found this in t/01sanity.t [7]

```
#!/usr/bin/perl -w

use Test::More tests => 7;

use Date::ICal;

# Make sure epoch time is being handled sanely.
my $t1 = Date::ICal->new( epoch => 0 );
is( $t1->epoch, 0,          "Epoch time of 0" );

# XXX This will only work on unix systems.
is( $t1->ical, '19700101Z', "  epoch to ical" );
is( $t1->year,  1970,       "  year()"  );
is( $t1->month, 1,          "  month()" );
is( $t1->day,   1,          "  day()"   );

# like the tests above, but starting with ical instead of epoch
my $t2 = Date::ICal->new( ical => '19700101Z' );
is( $t2->ical, '19700101Z', "Start of epoch in ICal notation" );
is( $t2->epoch, 0,          "  and back to ICal" );
```

The beginning of the epoch is different on most non-Unix operating systems [8]. Even

though Perl smooths out the differences for the most part, certain ports do it differently. MacPerl is one off the top of my head. [9] Rather than putting a comment in the test and hoping someone will read the test while debugging the failure, we can explicitly say it's never going to work and skip the test.

```
use Test::More tests => 7;
use Date::ICal;
# Make sure epoch time is being handled sanely.
my $t1 = Date::ICal->new( epoch => 0 );
is( $t1->epoch, 0,          "Epoch time of 0" );
SKIP: {
    skip('epoch to ICal not working on Mac OS', 6)
        if $^O eq 'MacOS';
    is( $t1->ical, '19700101Z', "  epoch to ical" );
    is( $t1->year,  1970,       "  year()"  );
    is( $t1->month, 1,          "  month()" );
    is( $t1->day,   1,          "  day()"   );
    # like the tests above, but starting with ical instead of epoch
    my $t2 = Date::ICal->new( ical => '19700101Z' );
    is( $t2->ical, '19700101Z', "Start of epoch in ICal notation" );
    is( $t2->epoch, 0,          "  and back to ICal" );
}
```

A little bit of magic happens here. When running on anything but MacOS, all the tests run normally. But when on MacOS, "skip()" causes the entire contents of the SKIP block to be jumped over. It never runs. Instead, "skip()" prints special output that tells Test::Harness that the tests have been skipped.

```
1..7
ok 1 - Epoch time of 0
ok 2 # skip epoch to ICal not working on MacOS
ok 3 # skip epoch to ICal not working on MacOS
ok 4 # skip epoch to ICal not working on MacOS
ok 5 # skip epoch to ICal not working on MacOS
ok 6 # skip epoch to ICal not working on MacOS
ok 7 # skip epoch to ICal not working on MacOS
```

This means your tests won't fail on MacOS. This means fewer emails from MacPerl users telling you about failing tests that you know will never work. You've got to be careful with skip tests. These are for tests which don't work and never will. It is not for skipping genuine bugs (we'll get to that in a moment).

The tests are wholly and completely skipped. [10] This will work.

```
SKIP: {
    skip("I don't wanna die!");
    die, die, die, die, die;
}
```

Todo tests

While thumbing through the Date::ICal man page, I came across this:

```
ical
    $ical_string = $ical->ical;
Retrieves, or sets, the date on the object, using any
valid ICal date/time string.
```

"Retrieves or sets". Hmmm. I didn't see a test for using "ical()" to set the date in the Date::ICal test suite. So I wrote one:

```
use Test::More tests => 1;
use Date::ICal;
my $ical = Date::ICal->new;
$ical->ical('20201231Z');
is( $ical->ical, '20201231Z',   'Setting via ical()' );
```

Run that. I saw:

```
1..1
not ok 1 - Setting via ical()
#     Failed test (- at line 6)
#          got: '20010814T233649Z'
#     expected: '20201231Z'
# Looks like you failed 1 tests of 1.
```

Whoops! Looks like it's unimplemented. Assume you don't have the time to fix this. [11] Normally, you'd just comment out the test and put a note in a todo list somewhere. Instead, explicitly state "this test will fail" by wrapping it in a "TODO" block:

```
use Test::More tests => 1;
```

```
    TODO: {
        local $TODO = 'ical($ical) not yet implemented';
        my $ical = Date::ICal->new;
        $ical->ical('20201231Z');
        is( $ical->ical, '20201231Z',   'Setting via ical()' );
    }
```

Now when you run, it's a little different:

```
    1..1
    not ok 1 - Setting via ical() # TODO ical($ical) not yet implemented
    #          got: '20010822T201551Z'
    #     expected: '20201231Z'
```

Test::More doesn't say "Looks like you failed 1 tests of 1". That '# TODO' tells Test::Harness "this is supposed to fail" and it treats a failure as a successful test.

You can write tests even before you've fixed the underlying code.

If a TODO test passes, Test::Harness will report it "UNEXPECTEDLY SUCCEEDED". When that happens, remove the TODO block with "local $TODO" and turn it into a real test.

Testing with taint mode.

Taint mode is a funny thing. It's the globalest of all global features. Once you turn it on, it affects all code in your program and all modules used (and all the modules they use). If a single piece of code isn't taint clean, the whole thing explodes. With that in mind, it's very important to ensure your module works under taint mode.

It's very simple to have your tests run under taint mode. Just throw a "-T" into the "#!" line. Test::Harness will read the switches in "#!" and use them to run your tests.

```
    #!/usr/bin/perl -Tw
    ...test normally here...
```

When you say "make test" it will run with taint mode on.

FOOTNOTES

1. The first number doesn't really mean anything, but it has to be 1. It's the second number that's important.
2. For those following along at home, I'm using version 1.31. It has some bugs, which is good -- we'll uncover them with our tests.
3. You can actually take this one step further and test the manual itself. Have a look at Test::Inline (formerly Pod::Tests).

4. Yes, there's a mistake in the test suite. What! Me, contrived?

5. We'll get to testing the contents of lists later.

6. But what happens if your test program dies halfway through?! Since we didn't say how many tests we're going to run, how can we know it failed? No problem, Test::More employs some magic to catch that death and turn the test into a failure, even if every test passed up to that point.

7. I cleaned it up a little.

8. Most Operating Systems record time as the number of seconds since a certain date. This date is the beginning of the epoch. Unix's starts at midnight January 1st, 1970 GMT.

9. MacOS's epoch is midnight January 1st, 1904. VMS's is midnight, November 17th, 1858, but vmsperl emulates the Unix epoch so it's not a problem.

10. As long as the code inside the SKIP block at least compiles. Please don't ask how. No, it's not a filter.

11. Do NOT be tempted to use TODO tests as a way to avoid fixing simple bugs!

AUTHORS

Michael G Schwern <schwern@pobox.com> and the perl-qa dancers!

MAINTAINERS

Chad Granum <exodist@cpan.org>

COPYRIGHT