Testworks User Guide

Release 1.0

Dylan Hackers

Sep 04, 2020

CONTENTS

CHAPTER

ONE

TESTWORKS USAGE

Contents

Testworks is a Dylan unit testing library.

See also: *Testworks Reference*

1.1 Quick Start

For the impatient, this section summarizes most of what you need to know to use Testworks.

Add `use testworks;` to both your test library and test module.

Tests contain arbitrary code and at least one assertion:

```
define test test-fn1 ()
  let v = do-something();
  assert-equal(fn1(v), "expected-value");
  assert-equal(fn1(v, key: 7), "seven", "regression test for bug/12345");
end;
```

If there are no assertions in a test it is considered "not implemented", which is displayed in the output (as a reminder to implement it) but is not considered a failure.

See also: *assert-true*, *assert-false*, *assert-signals*, and *assert-no-errors*. Each of these takes an optional *description* argument, which can be used to indicate the intent of the assertion if it isn't clear.

Benchmarks do not require any assertions and are automatically given the "benchmark" tag:

```
// Benchmark fn1
define benchmark fn1-benchmark ()
  fn1()
end;
```

See also, *benchmark-repeat*.

If you have a large or complex test library, "suites" may be used to organize tests into groups (for example one suite per module) and may be nested arbitrarily.

```
define suite my-library-suite ()
  suite module1-suite;
  suite module2-suite;
  test some-other-test;
end;
```

Note: Suites must be defined textually *after* the other suites and tests they contain.

To run your tests of course you need an executable and there are two ways to accomplish this:

1. Have your library call *run-test-application* and compile it as an executable. With no arguments *run-test-application* runs all tests and benchmarks, as filtered by the Testworks command-line options.

   If you prefer to manually organize your tests with suites, pass your top-level suite to *run-test-application* and that determines the initial set of tests that are filtered by the command line. **Note:** if you forget to add a test to any suite, the test will not be run.

1. Compile your test library as a shared library and run it with the `testworks-run` application. For example, for the *foo-test* library:

   ```
   _build/bin/testworks-run --load libfoo-test.so
   ```

In both cases *run-test-application* parses the command line so the options are the same. Use `--help` to see all options.

See *Suites* for a way to organize large test suites.

1.2 Defining Tests

1.2.1 Assertions

An assertion accepts an expression to evaluate and report back on, saying if the expression passed, failed, or signaled an error. As an example, in

```
assert-true(foo > bar)
```

the expression `foo > bar` is compared to `#f`, and the result is recorded by the test harness. Failing (or crashing) assertions do not cause the test to terminate; all assertions are run unless the test itself signals an error. (**NOTE:** See https://github.com/dylan-lang/testworks/issues/86 for plans to change this behavior.)

See the *Testworks Reference* for detailed documentation on the available assertion macros:

- *assert-true*
- *assert-false*
- *assert-equal*
- *assert-not-equal*
- *assert-signals*
- *assert-no-errors*
- *assert-instance?*
- *assert-not-instance?*

Each of these takes an optional description string, after the required arguments, which will be displayed if the assertion fails. If the description isn't provided, Testworks makes one from the expressions passed to the assertion macro. For example, `assert-true(2 > 3)` produces this failure message:

```
(2 > 3) is true failed [expression "(2 > 3)" evaluates to #f]
```

In general, Testworks should be pretty good at reporting the actual values that caused the failure so it shouldn't be necessary to include them in the description all the time.

In the future, there will be support for failures to include the source file line number for the assertion.

> *Note: You may also find check-* macros in Testworks test suites. These are a deprecated form of assertion. The only real difference between them and the assert-* macros is that they require a description of the assertion as the first argument.*

1.2.2 Tests

Tests contain assertions and arbitrary code needed to support those assertions. Each test may be part of a suite. Use the *test-definer* macro to define a test:

```
define test NAME (#key EXPECTED-FAILURE?, TAGS)
  BODY
end;
```

For example:

```
define test my-test ()
  assert-equal(2, 3);
  assert-equal(#f, #f);
  assert-true(identity(#t), "Check identity function");
end;
```

Note: if a test doesn't execute any assertions then it is marked as "not implemented" in the test results.

The result looks like this:

```
$ _build/bin/my-test
Running test my-test:
  2 = 3: [2 (from expression "2") and 3 (from expression "3") are not =.]
   FAILED in 0.000256s

my-test FAILED in 0.000256 seconds:
  Ran 0 suites: 0 passed (100.00000%), 0 failed, 0 skipped, 0 not implemented, 0␣
↪crashed
```

(continues on next page)

(continued from previous page)

```
  Ran 1 test: 0 passed (0.0%), 1 failed, 0 skipped, 0 not implemented, 0 crashed
  Ran 0 benchmarks: 0 passed (0.0%), 0 failed, 0 skipped, 0 not implemented, 0 crashed
  Ran 3 checks: 2 passed (66.666672%), 1 failed, 0 skipped, 0 not implemented, 0␣
↪crashed
```

Tests may be tagged with arbitrary strings, providing a way to select or filter out tests to run:

```
define test my-test-2 (tags: #["huge"])
  ...huge test that takes a long time...
end test;

define test my-test-3 (tags: #["huge", "verbose"])
  ...test with lots of output...
end test;
```

Tags can then be passed on the Testworks command-line. For example, this skips both of the above tests:

```
$ _build/bin/my-test-suite-app --tag=-huge --tag=-verbose
```

Negative tags take precedence, so `--tag=huge --tag=-verbose` runs `my-test-2` and skips `my-test-3`.

If the test is expected to fail, or fails under some conditions, Testworks can be made aware of this:

```
define test failing-test
    (expected-to-fail-reason: "bug 1234")
  assert-true(#f);
end test;

define test fails-on-windows
    (expected-to-fail?: method () $os-name = #"win32" end,
     expected-to-fail-reason: "blah is not implemented for WIN32 platform")
  if ($os-name = #"win32")
    assert-false(#t);
  else
    assert-true(#t);
  end if;
end test;
```

A test that is expected to fail and then fails is considered to be a passing test. If the test succeeds unexpectedly, it is considered a failing test. `expected-to-fail-reason:` **must** be supplied if `expected-to-fail?:` is true. An example of a good reason is a bug URL or other bug reference.

Test setup and teardown is accomplished with normal Dylan code using `block () ... cleanup ... end;`...

```
define test foo ()
  block ()
    do-setup-stuff();
    assert-equal(...);
    assert-equal(...);
  cleanup
    do-teardown-stuff()
  end
end;
```

1.2.3 Benchmarks

Benchmarks are like tests except for:

- They do not require any assertions. (They pass unless they signal an error.)
- They are automatically assigned the "benchmark" tag.

The *benchmark-definer* macro is like *test-definer*:

```
define benchmark my-benchmark ()
  ...body...
end;
```

Benchmarks may be added to suites:

```
define suite my-benchmarks-suite ()
  benchmark my-benchmark;
end;
```

Benchmarks and tests may be combined in the same suite. If you do that, tags may be used to run only the benchmarks (with `--tag=benchmark`) or only the tests (with `--tag=-benchmark`). If you are using suites anyway, you may wish to put benchmarks into a suite of their own. Example:

```
define suite strings-tests () ...only tests... end;
define suite strings-benchmarks () ...only benchmarks... end;
define suite strings-test-suite ()
  suite strings-tests;
  suite strings-benchmarks;
end;
```

See also, *benchmark-repeat*.

1.2.4 Suites

Suites are an optional feature that may be used to organize your tests into a hierarchy. Suites contain tests, benchmarks, and other suites. A suite is defined with the *suite-definer* macro. The format is:

```
define suite NAME (#key setup-function, cleanup-function)
    test TEST-NAME;
    benchmark BENCHMARK-NAME;
    suite SUITE-NAME;
end;
```

For example:

```
define suite first-suite ()
  test my-test;
  test example-test;
  test my-test-2;
  benchmark my-benchmark;
end;

define suite second-suite ()
  suite first-suite;
  test my-test;
end;
```

Suites can specify setup and cleanup functions via the keyword arguments `setup-function` and `cleanup-function`. These can be used for things like establishing database connections, initializing sockets and so on.

A simple example of doing this can be seen in the http-server test suite:

```
define suite http-test-suite (setup-function: start-sockets)
  suite http-server-test-suite;
  suite http-client-test-suite;
end;
```

Suites can be run via *run-test-application*. It should be called as the main function in an executable and will parse command-line args, execute tests and benchmarks, and generate reports. See the next section for details.

1.2.5 Interface Specification Suites

The *interface-specification-suite-definer* macro creates a normal test suite, much like `define suite` does, but based on an interface specification. For example,

```
define interface-specification-suite time-specification-suite ()
  sealed instantiable class <time> (<object>);
  constant $utc :: <zone>;
  variable *zone* :: <zone>;
  sealed generic function in-zone (<time>, <zone>) => (<time>);
  function now (#"key", #"zone") => (<time>);
  ...
end;
```

The specification usually has one clause, or "spec", for each name exported from your public interface module. Each spec creates a test named `test-{name}-specification` to verify that the implementation matches the spec for `{name}`. For example, by checking that the names are bound, that their bindings have the correct types, that functions accept the right number and types of arguments, etc.

Specification suites are otherwise just normal suites. They may include other arbitrary tests and child suites if desired:

```
define interface-specification-suite time-suite ()
  ...
  test test-time-still-moving-forward;
  suite time-travel-test-suite;
end;
```

This also means that if your interface is large you may use multiple *interface-specification-suite-definer* forms and then group them together.

See *interface-specification-suite-definer* for more details on the various kinds of specs.

1.3 Organizing Tests for One Library

If you don't use suites, the only organization you need is to name your tests and benchmarks uniquely, and you can safely skip the rest of this section. If you do use suites, read on....

Tests are used to combine related assertions into a unit, and suites further organize related tests and benchmarks. Suites may also contain other suites.

It is common for the test suite for library xxx to export a single test suite named xxx-test-suite, which is further subdivided into sub-suites, tests, and benchmarks as appropriate for that library. Some suites may be exported so that

they can be included as a component suite in combined test suites that cover multiple related libraries. (The alternative to this approach is running each library's tests as a separate executable.)

Note: It is an error for a test to be included in a suite multiple times, even transitively. Doing so would result in a misleading pass/fail ratio, and it is more likely to be a mistake than to be intentional.

The overall structure of a test library that is intended to be included in a combined test library may look something like this:

```
// --- library.dylan ---

define library xxx-tests
  use common-dylan;
  use testworks;
  use xxx;                  // the library you are testing
  export xxx-tests;         // so other test libs can include it
end;

define module xxx-tests
  use common-dylan;
  use testworks;
  use xxx;                  // the module you are testing
  export xxx-test-suite;    // so other suites can include it
end;

// --- main.dylan ---

define test my-awesome-test ()
  assert-true(...);
  assert-equal(...);
  ...
end;

define benchmark my-awesome-benchmark ()
  awesomely-slow-function();
end;

define suite xxx-test-suite ()
  test my-awesome-test;
  benchmark my-awesome-benchmark;
  suite my-awesome-other-suite;
  ...
end;
```

1.4 Running Your Tests As A Stand-alone Application

If you don't need to export any suites so they can be included in a higher-level combined test suite library (i.e., if you're happy running your test suite library as an executable) then you can simply call `run-test-application` to parse the standard testworks command-line options and run the specified tests:

```
run-test-application();          // if not using suites
run-test-application(my-suite);  // if using suites
```

and you can skip the rest of this section.

If you need to export a suite for use by another library, then you must also define a separate executable library, traditionally named "xxx-test-suite-app", which calls `run-test-application(xxx-test-suite)`.

Here's an example of such an application library:

1. The file `library.dylan` which must use at least the library that exports the test suite, and `testworks`:

```
Module:    dylan-user
Synopsis:  An application library for xxx-test-suite

define library xxx-test-suite-app
  use xxx-test-suite;
  use testworks;
end;

define module xxx-test-suite-app
  use xxx-test-suite;
  use testworks;
end;
```

2. The file `xxx-test-suite-app.dylan` which simply contains a call to the method *`run-test-application`* with the suite-name as an argument:

```
Module: xxx-test-suite-app

run-test-application(xxx-test-suite);
```

3. The file `xxx-test-suite-app.lid` which specifies the names of the source files:

```
Library: xxx-test-suite-app
Target-type: executable
Files: library.dylan
       xxx-test-suite-app.dylan
```

Once a library has been defined in this fashion it can be compiled into an executable with `dylan-compiler -build xxx-test-suite-app.lid` and run with `xxx-test-suite-app --help`.

1.5 Reports

The `--report` and `--report-file` options can be used to write a full report of test run results so that those results can be compared with subsequent test runs, for example to find regressions. These are the available report types:

failures Prints out only the list of failures and a summary.

json Outputs JSON objects that match the suite/test/assertion tree structure, with full detail.

summary (the default) Prints out only a summary of how many assertions, tests and suites were executed, passed, failed or crashed.

surefire Outputs XML in Surefire format. This elides information about specific assertions. This format is supported by various tools such as Jenkins.

xml Outputs XML that directly matches the suite/test/assertion tree structure, with full detail.

1.6 Comparing Test Results

*** To be filled in ***

Quick version:

- (master branch)$ my-test-suite –report json –report-file out1.json
- (your branch)$ my-test-suite –report json –report-file out2.json
- $ testworks-report out1.json out2.json

CHAPTER

TWO

TESTWORKS REFERENCE

Contents

See also: *Testworks Usage*

2.1 The Testworks Module

2.1.1 Suites, Tests, and Benchmarks

test-definer Macro

Define a new test.

Signature define test *test-name* (#key *expected-to-fail?, expected-to-fail-reason, tags*) *body* end

Parameters

- **test-name** – Name of the test; a Dylan variable name.
- **expected-to-fail?** (*#key*) – An instance of either <boolean> or <function>. This indicates whether or not the test is expected to fail.
- **expected-to-fail-reason** (*#key*) – A <string> or #f. Must be supplied if expected-to-fail? is true. A good reason usually references a bug.
- **tags** (*#key*) – A list of strings to tag this test.

Tests may contain arbitrary code, plus any number of assertions. If any assertion fails the test will fail, but any remaining assertions in the test will still be executed. If code outside of an assertion signals an error, the test is marked as "crashed" and remaining assertions are skipped.

If *expected-to-fail?* is set to #t or a function that when executed returns a true value, then the test will be expected to fail. Such a failure is treated as a successful test run. If the test passes rather than failing, it is considered a test failure. This option has no effect on tests which are *not implemented* or which have *crashed*.

expected-to-fail-reason is required if the test is expected to fail. Normally it should reference a bug (a URL or at least a bug number). If *expected-to-fail-reason* is supplied, *expected-to-fail?* may be omitted because it is implied to be `#t`.

tags provide a way to select or filter out specific tests during a test run. The Testworks command-line (provided by *run-test-application*) has a `--tag` option to only run tests that match (or don't match) specific tags.

benchmark-definer Macro

Define a new benchmark.

Signature define benchmark *name* (#key *expected-to-fail?, tags*) *body* end

Parameters

- **name** – Name of the benchmark; a Dylan variable name.
- **expected-to-fail?** (*#key*) – An instance of either <boolean> or <function>. This indicates whether or not the test is expected to fail.
- **tags** (*#key*) – A list of strings to tag this benchmark.

Benchmarks may contain arbitrary code and do not require any assertions. If the benchmark signals an error it is marked as "crashed". Other than this, and some differences in how the results are displayed, benchmarks are the same as tests.

benchmark-repeat Macro

Repeatedly execute a block of code, recording profiling information for each execution.

Signature benchmark-repeat (#key *iterations* = 1) *body* end

Parameters

- **iterations** – Number of times to execute *body*.

Results for benchmarks that call benchmark-repeat display the min, max, mean, and median run times across all iterations.

It may be necessary to use `--report=full` to display detailed benchmark statistics.

At the beginning of each iteration benchmark-repeat first collects garbage to attempt to reduce variability across different executions.

suite-definer Macro

Define a new test suite.

Signature define suite *suite-name* (#key *setup-function cleanup-function*) *body* end

Parameters

- **suite-name** – Name of the suite; a Dylan variable name.
- **setup-function** (*#key*) – A function to perform setup before the suite starts.
- **cleanup-function** (*#key*) – A function to perform teardown after the suite finishes.

Suites provide a way to group tests and other suites into a single executable unit. Suites may be nested arbitrarily.

setup-function is executed before any tests or sub-suites are run. If *setup-function* signals an error the entire suite is skipped and marked as "crashed".

cleanup-function is executed after all sub-suites and tests have completed, regardless of whether an error is signaled.

interface-specification-suite-definer Macro

Define a test suite to verify an API.

Signature define interface-specification-suite *suite-name* () *specs* end;

Parameters

- **suite-name** – Name of the suite; a Dylan variable name.

This macro is useful to verify that public interfaces to your library don't change unintentionally.

specs are clauses separated by semicolons, specifying the attributes of an exported name. Each *spec* looks much like the definition of the name being tested. The following example has one of each kind of spec:

```
define interface-specification-suite time-specification-suite ()
  sealed instantiable abstract class <time> (<object>);
  generic function parse-time (<string>, #"key") => (<time>);
  variable *foo* :: <string>;
  constant $unix-epoch :: <time>;
end;
```

The following sections explain the syntax of each kind of spec in detail. Note that there is no way to verify macros automatically and therefore there is no "macro" spec.

class specs

Syntax: *modifiers* class *name* (*superclasses*) [, *test-options*];

modifiers

`sealed` or `open`, `primary` or `free`, `abstract` or `concrete`, and `instantiable`. Currently the first two pairs are unused, but you may want to specify them anyway, to keep the spec in sync with the code.

If `instantiable` is specified, Testworks will try to make an instance of *name* by calling `make` with no arguments. If your class requires init arguments, you must define a method on `make-test-instance`:

```
define method make-test-instance
    (class == <my-class>) => (instance :: <my-class>)
  make(<my-class>, ...init args...)
end
```

name

Name of the class to verify.

superclasses

Comma-separated list of superclass names.

test-options

Any options valid for *test-definer*. For example, `expected-to-fail-reason: "foo"`.

function specs

Syntax: *modifiers* function *name* (*parameter-types*) => (*value-types*) [, *test-options*];

modifiers

`generic`

name

Name of the function. Note that function specs should be used for functions created with `define function` (which are really just bare methods bound to a name as with `define constant m = method() ... end`) and for generic functions.

parameter-types

Comma-separated list of parameter type names, possibly empty. Where `#rest`, `#key`, and `#all-keys` appear in the corresponding function definition, use `#"rest"`, `#"key"`, and `#"all-keys"` instead (i.e., with double quotes). Keyword arguments are specified *without* type qualifiers. Examples from the dylan-test-suite:

```
open generic function make
    (<type>, #"rest", #"key", #"all-keys") => (<object>);
open generic function copy-sequence
    (<sequence>, #"key", #"start", #"end") => (<sequence>);
```

value-types

Comma-separated list of return value type names, possibly empty.

test-options

Any options valid for *test-definer*. For example, `expected-to-fail-reason: "foo"`.

variable specs

Syntax: variable *name* :: *type* [, *test-options*];

name

Name of the variable.

type

Type of the variable.

test-options

Any options valid for *test-definer*. For example, `expected-to-fail-reason: "foo"`.

constant specs

Syntax: constant *name* :: *type* [, *test-options*];

name

Name of the constant.

type

Type of the constant.

test-options

Any options valid for *test-definer*. For example, `expected-to-fail-reason: "foo"`.

2.1.2 Assertions

Assertions are the smallest unit of verification in Testworks. They must appear within the body of a test.

Assertion macros that accept an argument that is the expected value as well as the expression that is to be tested typically expect the value first and the expression second. The macros don't always require that this be the case:

```
assert-not-equal(5, 2 + 2);
assert-instance?(<integer>, 2 + 2);
```

All assertion macros accept a description of what is being tested as an *optional* final argument. The description should be stated in the positive sense. For example:

```
assert-equal(2, 2 + 2, "2 + 2 equals 2")
```

These are the available assertion macros:

- *assert-true*
- *assert-false*
- *assert-equal*
- *assert-not-equal*
- *assert-signals*
- *assert-no-errors*
- *assert-instance?*
- *assert-not-instance?*

assert-true Macro

Assert that an expression evaluates to a true value. Importantly, this does not mean the expression is exactly `#t`, but rather that it is *not* `#f`. If you want to explicitly test for equality to `#t` use `assert-equal(#t, ...)` or `assert-true(#t = ...)`.

Signature assert-true *expression* [*description*]

Parameters

- **expression** – any expression
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "two is less than three". If no description is supplied one will be automatically generated based on the text of the expression.

Example

```
assert-true(has-fleas?(my-dog))
assert-true(has-fleas?(my-dog), "my dog has fleas")
```

assert-false Macro

Assert that an expression evaluates to `#f`.

Signature assert-false *expression* [*description*]

Parameters

- **expression** – any expression
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "three is less than two". If no description is supplied one will be automatically generated based on the text of the expression.

Example

```
assert-false(3 < 2)
assert-false(6 = 7, "six equals seven")
```

assert-equal Macro

Assert that two values are equal using = as the comparison function. Using this macro is preferable to using `assert-true(a = b)` because the failure messages are much better when comparing certain types of objects, such as collections.

Signature assert-equal *expression1 expression2* [*description*]

Parameters

- **expression1** – any expression
- **expression2** – any expression
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "two equals two". If no description is supplied one will be automatically generated based on the text of the two expressions.

Example

```
assert-equal(2, my-complicated-method())
assert-equal(this, that, "this and that are the same")
```

assert-not-equal Macro

Assert that two values are not equal using ~= as the comparison function. Using this macro is preferable to using `assert-true(a ~= b)` or `assert-false(a = b)` because the generated failure messages can be better.

Signature assert-not-equal *expression1 expression2* [*description*]

Parameters

- **expression1** – any expression
- **expression2** – any expression
- **description** – A description of what the assertion tests. This should be stated so as to express what the correct result would be, for example "two does not equal three". If no description is supplied one will be automatically generated based on the text of the two expressions.

Example

```
assert-not-equal(2, my-complicated-method())
assert-not-equal(this, that, "this does not equal that")
```

assert-signals Macro

Assert that an expression signals a given condition class.

Signature assert-signals *condition*, *expression* [*description*]

Parameters

- **condition** – an expression that yields a condition class
- **expression** – any expression
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "two is less than three". If no description is supplied one will be automatically generated based on the text of the expression.

The assertion succeeds if the expected *condition* is signaled by the evaluation of *expression*.

Example

```
assert-signals(<division-by-zero-error>, 3 / 0)
assert-signals(<division-by-zero-error>, 3 / 0,
               "my super special description")
```

assert-no-errors Macro

Assert that an expression does not signal any errors.

Signature assert-no-errors *expression* [*description*]

Parameters

- **expression** – any expression
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "two is less than three". If no description is supplied one will be automatically generated based on the text of the expression.

The assertion succeeds if no error is signaled by the evaluation of *expression*.

Use of this macro is preferable to simply executing *expression* as part of the test body for two reasons. First, it can clarify the purpose of the test, by telling the reader "here's an expression that is explicitly being tested, and not just part of the test setup." Second, if the assertion signals an error the test will record that fact and continue, as opposed to taking a non-local exit. Third, it will show up in generated reports.

Example

```
assert-no-errors(my-hairy-logic())
assert-no-errors(my-hairy-logic(),
                 "hairy logic completes without error")
```

assert-instance? Macro

Assert that the result of an expression is an instance of a given type.

Signature assert-instance? *type expression* [*description*]

Parameters

- **type** – The expected type.
- **expression** – An expression.
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "two is less than three". If no description is supplied one will be automatically generated based on the text of the expression.

Discussion

> **Warning:** The arguments to this assertion follow the typical argument ordering of Testworks assertions with the desired value before the expression that represents the test. As such, the desired *type* is the first parameter to this assertion while it is the second parameter for `instance?`.

Example

```
assert-instance?(<type>, subclass(<string>));

assert-instance?(<type>, subclass(<string>),
                 "subclass returns type");
```

assert-not-instance? Macro

Assert that the result of an expression is **not** an instance of a given class.

Signature assert-not-instance? *type expression* [*description*]

Parameters

- **type** – The type.
- **expression** – An expression.
- **description** – A description of what the assertion tests. This should be stated in positive form, such as "two is less than three". If no description is supplied one will be automatically generated based on the text of the expression.

Discussion

> **Warning:** The arguments to this assertion follow the typical argument ordering of Testworks assertions with the desired value before the expression that represents the test. As such, the desired *type* is the first parameter to this assertion while it is the second parameter for `instance?`.

Example

```
assert-not-instance?(limited(<integer>, min: 0), -1);

assert-not-instance?(limited(<integer>, min: 0), -1,
                     "values below lower bound are not instances");
```

2.1.3 Checks

Checks are deprecated; use *Assertions* instead. The main difference between checks and assertions is that the check macros require a description as their first argument, whereas assertions do not.

These are the available checks:

- *check*
- *check-true*
- *check-false*
- *check-equal*
- *check-instance?*
- *check-condition*

check Macro

Perform a check within a test.

Signature check *name function* #rest *arguments*

Parameters

- **name** – An instance of `<string>`.
- **function** – The function to check.
- **arguments** (*#rest*) – The arguments for `function`.

Example

```
check("Test less than operator", \<, 2, 3)
```

check-condition Macro

Check that a given condition is signalled.

Signature check-condition *name expected expression*

Parameters

- **name** – An instance of `<string>`.
- **expected** – The expected condition class.
- **expression** – An expression.

Example

```
check-condition("format-to-string crashes when missing an argument",
                <error>, format-to-string("Hello %s"));
```

check-equal Macro

Check that 2 expressions are equal.

Signature check-equal *name expected expression*

Parameters

- **name** – An instance of `<string>`.
- **expected** – The expected value of `expression`.
- **expression** – An expression.

Example

```
check-equal("condition-to-string of an error produces correct string",
            "Hello",
            condition-to-string(make(<simple-error>, format-string:
↪"Hello")));
```

check-false Macro

Check that an expression has a result of `#f`.

Signature check-false *name expression*

Parameters

- **name** – An instance of `<string>`.
- **expression** – An expression.

Example

```
check-false("unsupplied?(#f) == #f", unsupplied?(#f));
```

check-instance? Macro

Check that the result of an expression is an instance of a given type.

Signature check-instance? *name type expression*

Parameters

- **name** – An instance of <string>.
- **type** – The expected type.
- **expression** – An expression.

Example

```
check-instance?("subclass returns type",
                <type>, subclass(<string>));
```

check-true Macro

Check that the result of an expression is not #f.

Signature check-true *name expression*

Parameters

- **name** – An instance of <string>.
- **expression** – An expression.

Discussion Note that if you want to explicitly check if an expression evaluates to #t, you should use *check-equal*.

Example

```
check-true("unsupplied?($unsupplied)", unsupplied?($unsupplied));
```

2.1.4 Test Execution

run-test-application Function

Run a test suite or test as part of a stand-alone test executable.

Signature run-test-application #rest *suite-or-test* => ()

Parameters

- **suite-or-test** – (optional) An instance of <suite> or <runnable>. If not supplied then all tests and benchmarks are run.

This is the main entry point to run a set of tests in Testworks. It parses the command-line and based on the specified options selects the set of suites or tests to run, runs them, and generates a final report of the results.

Internally, *run-test-application* creates a <test-runner> based on the command-line options and then calls run-tests with the runner and *suite-or-test*.

test-option Function

Return an option value passed on the test-application command line.

Signature test-option *name* #key *default* => *value*

Parameters

- **name** – An instance of type <string>.
- **default** (*#key*) – An instance of type <string>.

Values

- **value** – An instance of type <string>.

Returns an option value passed to the test on the test application command line, in the form `*name*=*value*`. If no option value was given, the *default* value is returned if one was supplied, otherwise an error is signalled.

This feature allows information about external resources, such as path names of reference data files, or the hostname of a test database server, to be supplied on the command line of the test application and retrieved by the test.

`test-temp-directory` Function

Retrieve a unique temporary directory for the current test to use.

Signature test-temp-directory => (directory :: <directory-locator>)

Returns a directory (a `<directory-locator>`) that may be used for temporary files created by the test or benchmark. The directory is created the first time this function is called for each test or benchmark and is not deleted after the test run is complete in case it's useful for post-mortem analysis. The directory is named `_test/<user>-<timestamp>/<test-name>` and is rooted at `$DYLAN`, if defined, or in the current directory otherwise.

CHAPTER

THREE

COPYRIGHT

In particular, large parts of this documentation resulted from the work of Judy Anderson, Shri Amit and Andy Armstrong at Harlequin.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

API INDEX

INDEX