rules_testing

Release 0.0.0

Bazel

Feb 20, 2024

CONTENTS

rules_testing is a collection of utilities, libraries, and frameworks to make testing Starlark and Bazel rules easy and pleasant.

version |version|

CHAPTER

ONE

INSTALLATION

To use rules_testing, you need to modify `WORKSPACE` or `MODULE.bazel` to depend on rules_testing. We recommend using bzlmod because it's simpler.

For bzlmod, add this to your `MODULE.bazel`:

```
bazel_dep(name = "rules_testing", version = "<VERSION>", dev_dependency=True)
```

See the GitHub releases page for available versions.

For `WORKSPACE`, see the GitHub releases page for the necessary config to copy and paste.

CHAPTER

TWO

ANALYSIS TESTS

Analysis testing means testing something during the analysis phase of Bazel execution – this is when rule logic is run. See *Analysis tests* for how to write analysis tests.

CHAPTER

THREE

FLUENT ASSERTS

Included in rules_testing is a fluent, truth-style asserts library.

See *Truth docs* for how to use it.

3.1 Analysis Tests

Analysis tests are the typical way to test rule behavior. They allow observing behavior about a rule that isn't visible to a regular test as well as modifying Bazel configuration state to test rule behavior for e.g. different platforms.

If you've ever wanted to verify…

- A certain combination of flags
- Building for another OS
- That certain providers are returned
- That aspects behaved a certain way

Or other observable information, then an analysis test does that.

3.1.1 Quick start

For a quick copy/paste start, create a `.bzl` file with your test code, and a `BUILD.bazel` file to load your tests and declare them. Here's a skeleton:

```
# BUILD
load(":my_tests.bzl", "my_test_suite")

my_test_suite(name="my_test_suite")
```

```
# my_tests.bzl

load("@rules_testing//lib:analysis_test.bzl", "test_suite", "analysis_test")
load("@rules_testing//lib:util.bzl", "util")

def _test_hello(name):
    util.helper_target(
        native.filegroup,
        name = name + "_subject",
        srcs = ["hello_world.txt"],
```

(continues on next page)

(continued from previous page)

```
    )
    analysis_test(
        name = name,
        impl = _test_hello_impl,
        target = name + "_subject"
    )

def _test_hello_impl(env, target):
    env.expect.that_target(target).default_outputs().contains(
        "hello_world.txt"
    )

def my_test_suite(name):
    test_suite(
        name = name,
        tests = [
            _test_hello,
        ]
    )
```

3.1.2 Arranging the test

The arrange part of a test defines a target using the rule under test and sets up its dependencies. This is done by writing a macro, which runs during the loading phase, that instantiates the target under test and dependencies. All the targets taking part in the arrangement should be tagged with `manual` so that they are ignored by common build patterns (e.g. `//...` or `foo:all`).

Example:

```
load("@rules_proto/defs:proto_library.bzl", "proto_library")


def _test_basic(name):
  """Verifies basic behavior of a proto_library rule."""
  # (1) Arrange
  proto_library(name=name + '_foo', srcs=["foo.proto"], deps=[name + "_bar"], tags=[
↪"manual"])
  proto_library(name=name + '_bar', srcs=["bar.proto"], tags=["manual"])

  # (2) Act
  ...
```

TIP: Source source files aren't required to exist. This is because the analysis phase only records the path to source files; they aren't read until after the analysis phase. The macro function should be named after the behaviour being tested (e.g. `_test_frob_compiler_passed_qux_flag`). The setup targets should follow the macro naming conventions, that is all targets should include the name argument as a prefix – this helps tests avoid creating conflicting names.

Limitations

Bazel limits the number of transitive dependencies that can be used in the setup. The limit is controlled by `--analysis_testing_deps_limit` flag.

Mocking toolchains (adding a toolchain used only in the test) is not possible at the moment.

3.1.3 Running the analysis phase

The act part runs the analysis phase for a specific target and calls a user supplied function. All of the work is done by Bazel and the framework. Use `analysis_test` macro to pass in the target to analyse and a function that will be called with the analysis results:

```
load("@rules_testing//lib:analysis_test.bzl", "analysis_test")


def _test_basic(name):
  ...

  # (2) Act
  analysis_test(name, target=name + "_foo", impl=_test_basic)
```

3.1.4 Assertions

The assert function (in example `_test_basic`) gets `env` and `target` as parameters, where...

- `env` is information about the overall build and test
- `target` is the target under test (as specified in the `target` attribute during the arrange step).

The `env.expect` attribute provides a `truth.Expect` object, which allows writing fluent asserts:

```


def _test_basic(env, target):
  env.expect.assert_that(target).runfiles().contains_at_least("foo.txt")
  env.expect.assert_that(target).action_generating("foo.txt").contains_flag_values("--a")
```

Note that you aren't *required* to use `env.expect`. If you want to perform asserts another way, then `env.fail()` can be called to register any failures.

3.1.5 Collecting the tests together

Use the `test_suite` function to collect all tests together:

```
load("@rules_testing//lib:analysis_test.bzl", "test_suite")


def proto_library_test_suite(name):
  test_suite(
      name=name,
```

(continues on next page)

(continued from previous page)

```
      tests=[
          _test_basic,
          _test_advanced,
      ]
  )
```

In your BUILD file instantiate the suite:

```
load("//path/to/your/package:proto_library_tests.bzl", "proto_library_test_suite")
proto_library_test_suite(name = "proto_library_test_suite")
```

The function instantiates all test macros and wraps them into a single target. This removes the need to load and call each test separately in the BUILD file.

Advanced test collection, reuse, and parameterizing

If you have many tests and rules and need to re-use them between each other, then there are a couple tricks to make it easy:

- Tests aren't required to all be in the same file. So long as you can load the arrange function and pass it to `test_suite`, then you can split tests into multiple files for reuse.
- Similarly, arrange functions themselves aren't required to take only a `name` argument – only the functions passed to `test_suite.test` require this.

By using lists and lambdas, we can define collections of tests and have multiple rules reuse them:

```
# base_tests.bzl

_base_tests = []

def _test_common(name, rule_under_test):
  rule_under_test(...)
  analysis_test(...)

def _test_common_impl(env, target):
  env.expect.that_target(target).contains(...)

_base_tests.append(_test_common)

def create_base_tests(rule_under_test):
    return [
        lambda name: test(name=name, rule_under_test=rule_under_test)
        for test in _base_tests
    ]

# my_binary_tests.bzl
load("//my/my_binary.bzl", "my_binary")
load(":base_tests.bzl", "create_base_tests")
load("@rules_testing//lib:analysis_test.bzl", "test_suite")

def my_binary_suite(name):
    test_suite(
```

(continues on next page)

(continued from previous page)

```
        name = name,
        tests = create_base_tests(my_binary)
    )

# my_test_tests.bzl
load("//my/my_test.bzl", "my_test")
load(":base_tests.bzl", "base_tests")
load("@rules_testing//lib:analysis_test.bzl", "test_suite")

def my_test_suite(name):
    test_suite(
        name = name,
        tests = create_base_tests(my_test)
    )
```

3.1.6 Multiple targets under test

There's two ways that multiple targets can be tested in a single test: passing a list or dict to the `target` (singular) arg, or using the `targets` (plural) arg. The main difference is the `target` arg uses the same settings for each target. The `targets` arg, in comparison, allows you to customize the settings for each entry in the dict. Under the hood, the `target` arg is a single attribute, while the `targets` arg has a separate attribute for each dict entry.

Multiple targets with the same settings

If you need to compare multiple targets with the same settings, then the simple way to do this is to pass a list or map of targets to the `target` arg. An alternative is to use the `targets` arg (see below)

In the example below, we verify that changing the `bar` arg doesn't change the outputs generated.

```
def _test_multiple(name):
    foo_binary(name = name + "_bar_true", bar=True),
    foo_binary(name = name + "_bar_false", bar=False),
    analysis_test(
        name = name,
        target = [name + "_bar_true", name + "_bar_false"],
        impl = _test_multiple_impl,
)

def _test_multiple_impl(env, targets): # targets here is a list targets
    # Verify bar_true and bar_false have the same default outputs
    env.expect.that_target(
        targets[0]
    ).default_outputs().contains_exactly(
        targets[1][DefaultInfo].files
    )
```

Alternatively, the `targets` arg can be used, which is more useful if targets need different configurations or should have an associated name.

Multiple targets with different config settings

If you need to compare multiple targets, or verify the effect of different configurations on one or more targets, then this is possible by using the `targets` arg and defining custom attributes (`attrs` arg) using dictionaries with some special keys.

In the example below, the same target is built in two different configurations and then it's verified that they have the same runfiles.

```
def _test_multiple_configs(name):
    foo_binary(name = name + "_multi_configs"),
    analysis_test(
        name = name,
        targets = {
            "config_a": name + "_multi_configs",
            "config_b": name + "_multi_configs",
        },
        attrs = {
            "config_a": {
                "@config_settings": {"//foo:setting": "A"},
            },
            "config_b": {
                "@config_settings": {"//foo:setting": "B"},
            },
        },
        impl = _test_multiple_impl,
)

def _test_multiple_impl(env, targets):
    # Verify the same target under different configurations have equivalent
    # runfiles
    env.expect.that_target(
        targets.config_a
    ).default_runfiles().contains_exactly(
        runfiles_paths(env.ctx.workspace_name, targets.config_b.default_runfiles)
    )
```

Note that they don't have to have custom settings. If they aren't customized, they will just use the base target-under-test settings. This is useful if you want each target to have an named identifier for clarity.

Gotchas when comparing across configurations

- Generated files (`File` objects with `is_source=False`) include their configuration, so files built in different configurations cannot compare equal. This is true even if they would have identical content – recall that an analysis test doesn't read the content and only sees the metadata about files.

3.1.7 Custom attributes

Tests can have their attributes customized using the `attrs` arg. There are two different types of values that can be specified: dicts and attribute objects (e.g. `attr.string()` objects.

When an attribute object is given, it is used as-is. These are most useful for specifying implicit values the implementation function might need:

When a dict is given, it acts as a template that will have target-under-test settings mixed into it, e.g. config settings, aspects, etc. Attributes defined using this style are considered targets under test and will be passed as part of the `targets` arg to the implementation function.

```
analysis_test(
    name = "test_custom_attributes",
    targets = {
          "subject": ":subject",
    }
    attr_values = {
        "is_windows": select({
            "@plaforms//os:windows": True,
            "//conditions:default": False
        }),
    }
    attrs = {
      "is_windows": attr.bool(),
      "subject": {
          "@attr": attr.label,
          "@config_settings": {...}
          "aspects": [custom_aspect],
      }
    },
)
```

3.1.8 Tips and best practices

- Use private names for your tests, `def _test_foo`. This allows buildifier to detect when you've forgotten to put a test in the `tests` attribute. The framework will strip leading underscores from the test name
- Tag the arranged inputs of your tests with `tags=["manual"]`; the `util.helper_target` function helps with this. This prevents common build patterns (e.g. `bazel test //...` or `bazel test :all`) from trying to build them.
- Put each rule's tests into their own directory with their own BUILD file. This allows better isolation between the rules' test suites in several ways:
  - When reusing tests, target names are less likely to collide.
  - During the edit-run cycle, modifications to verify one rule that would break another rule can be ignored until you're ready to test the other rule.

3.2 Best Practices

Here we collection tips and techniques for keeping your tests maintainable and avoiding common pitfalls.

3.2.1 Put each suite of tests in its own sub-package

It's recommended to put a given suite of unit tests in their own sub-package (directory with a BUILD file). This is because the names of your test functions become target names in the BUILD file, which makes it easier to create name conflicts. By moving them into their own package, you don't have to worry about unit test function names in one `.bzl` file conflicting with names in another.

3.2.2 Give test functions private names

It's recommended to give test functions private names, i.e. start with a leading underscore. This is because if you forget to add a test to the list of tests (an easy mistake to make in a file with many tests), the test won't run, and it'll appear as if everything is OK. By using a leading underscore, tools like buildifier can detect the unused private function and will warn you that it's unused, preventing you from accidentally forgetting it.

3.3 Guides

3.3.1 Analysis tests

Analysis testing means testing something during the analysis phase of Bazel execution – this is when rule logic is run.

See *Analysis testing* for how to write analysis tests.

3.3.2 Fluent asserts

Included in rules_testing is a fluent, truth-style asserts library.

See *Truth docs* for how to use it.

3.4 Test suites

The `test_suite` macro is a front-end for easily instantiating groups of Starlark tests. It can handle both analysis tests and unit tests. Under the hood, each test is its own target with an aggregating `native.test_suite` for the group of tests.

3.4.1 Basic tests

Basic tests are tests that don't require any custom setup or attributes. This is the common case for tests of utility code that doesn't interact with objects only available to rules (e.g. Targets). These tests are created using `unit_test`.

To write such a test, simply write a `unit_test` compatible function (one that accepts `env`) and pass it to `test_suite.basic_tests`.

```
# BUILD

load(":my_tests.bzl", "my_test_suite")
load("@rules_testing//lib:test_suite.bzl", "test_suite")

my_test_suite(name = "my_tests")

# my_tests.bzl

def _foo_test(env):
  env.expect.that_str(...).equals(...)

def my_test_suite(name):
  test_suite(
    name = name,
    basic_tests = [
      _foo_test,
    ]
  )
```

Note that it isn't *required* to write a custom test suite function, but doing so is preferred because it's uncommon for BUILD files to pass around function objects, and tools won't be confused by it.

3.4.2 Regular tests

A regular test is a macro that acts as a setup function and is expected to create a target of the given name (which is added to the underlying test suite).

The setup function can perform arbitrary logic, but in the end, it's expected to call `unit_test` or `analysis_test` to create a target with the provided name.

If you're writing an `analysis_test`, then you're writing a regular test.

```
# my_tests.bzl
def _foo_test(name):
  analysis_test(
    name = name,
    impl = _foo_test_impl,
    attrs = {"myattr": attr.string(default="default")}
  )

def _foo_test_impl(env):
  env.expect.that_str(...).equals(...)

def my_test_suite(name):
  test_suite(
```

(continues on next page)

(continued from previous page)

```
    name = name,
    tests = [
      _foo_test,
    ]
  )
```

Note that a using a setup function with `unit_test` test is not required to define custom attributes; the above is just an example. If you want to define custom attributes for every test in a suite, the `test_kwargs` argument of `test_suite` can be used to pass additional arguments to all tests in the suite.

3.5 Truth Guide

Also see: *Truth API reference*

3.5.1 What is Truth?

Truth is a style of doing asserts that makes it easy to perform complex assertions that are easy to understand and give actionable error messages.

The basic way it works is wrapping a value in a type-specific object that provides type-specific assertion methods. This style provides several benefits:

- A fluent API that more directly expresses the assertion
- More egonomic assert functions
- Error messages with more informative context
- Promotes code reuses at the type-level.

3.5.2 Example Usage

Note that all examples assume usage of the rules_testing `analysis_test` framework, but truth itself does not require it.

```
def test_foo(env, target):
    subject = env.expect.that_target(target)
    subject.runfiles().contains_at_least(["foo.txt"])
    subject.executable().equals("bar.exe")

    subject = env.expect.that_action(...)
    subject.contains_at_least_args(...)
```

3.5.3 Subjects

Subjects are wrappers around a value that provide ways to assert on the value, access sub-values of it, or otherwise augment interacting with the wrapped value. For example, `TargetSubject` wraps Bazel `Target` objects and `RunfilesSubject` wraps Bazel `runfiles` objects. Normally accessing a target's runfiles and verifying the runfiles contents would require the verbose `target[DefaultInfo].default_runfiles`, plus additional code to convert a `runfiles` object's `files`, `symlinks`, `root_symlinks`, and `empty_filenames` into a single list to verify. With subject classes, however, it can be concisely expressed as `expect.that_target(target).runfiles().contains(path)`.

The Truth library provides subjects for types that are built into Bazel, but custom subjects can be implemented to handle custom providers or other objects.

3.5.4 Predicates

Because Starlark's data model doesn't allow customizing equality checking, some subjects allow matching values by using a predicate function. This makes it easier to, for example, ignore a platform-specific file extension.

This is implemented using the structural `Matcher` "interface". This is a struct that contains the predicate function and a description of what the function does, which allows for more intelligible error messages.

A variety of matchers are in `truth.bzl#matching`, but custom matches can be implemented using `matching.custom_matcher`

3.5.5 Writing a new Subject

Writing a new Subject involves two basic pieces:

1. Creating a constructor function, e.g. `_foo_subject_new`, that takes the actual value and an `ExpectMeta` object (see `_expect_meta_new()`).
2. Adding a method to `expect` or another Subject class to pass along state and instantiate the new subject; both may be modified if the actual object can be independenly created or obtained through another subject.

   For top-level subjects, a method named `that_foo()` should be added to the `expect` class.

   For child-subjects, an appropriately named method should be added to the parent subject, and the parent subject should call `ExpectMeta.derive()` to create a new set of meta data for the child subject.

The assert methods a subject provides are up to the subject, but try to follow the naming scheme of other subjects. The purpose of a custom subject is to make it easier to write tests that are correct and informative. It's common to have a combination of ergonomic asserts for common cases, and delegating to child-subjects for other cases.

3.5.6 Adding asserts to a subject

Fundamentally, an assert method calls `ExpectMeta.add_failure()` to record when there is a failure. That method will wire together any surrounding context with the provided error message information. Otherwise an assertion is free to implement checks how it pleases.

The naming of functions should mostly read naturally, but doesn't need to be perfect grammatically. Be aware of ambiguous words like "contains" or "matches". For example, `contains_flag("--foo")` – does this check that "–flag" was specified at all (ignoring value), or that it was specified and has no value?

Assertion functions can make use of a variety of helper methods in processing values, comparing them, and generating error messages. Helpers of particular note are:

- `_check_*`: These functions implement comparison, error formatting, and error reporting.

- `_compare_*`: These functions implements comparison for different cases and take care of various edge cases.
- `_format_failure_*`: These functions create human-friendly messages describing both the observed values and the problem with them.
- `_format_problem_*`: These functions format only the problem identified.
- `_format_actual_*`: These functions format only the observed values.

3.6 Unit tests

Unit tests are for Starlark code that isn't specific to analysis-phase or loading phase cases; usually utility code of some sort. Such tests typically don't require a rule `ctx` or instantiating other targets to verify the code under test.

To write such a test, simply write a function accepting `env` and pass it to `test_suite`. The test suite will pass your verification function to `unit_test()` for you.

```
# BUILD

load(":my_tests.bzl", "my_test_suite")
load("@rules_testing//lib:test_suite.bzl", "test_suite")

my_test_suite(name = "my_tests")

# my_tests.bzl

def _foo_test(env):
  env.expect.that_str(...).equals(...)

def my_test_suite(name):
  test_suite(
    name = name,
    basic_tests = [
      _foo_test,
    ]
  )
```

Note that it isn't *required* to write a custom test suite function, but doing so is preferred because it's uncommon for BUILD files to pass around function objects, and tools won't be confused by it.

3.6.1 Customizing setup

If you want to customize the setup (loading phase) of a unit test, e.g. to add custom attributes, then you need to write in the same style as an analysis test: one function is a verification function, and another function performs setup and calls `unit_test()`, passing in the verification function.

Custom tests are like basic tests, except you can hook into the loading phase before the actual unit test is defined. Because you control the invocation of `unit_test`, you can e.g. define custom attributes specific to the test.

```
# my_tests.bzl
def _foo_test(name):
  unit_test(
    name = name,
```

(continues on next page)

(continued from previous page)

```
    impl = _foo_test_impl,
    attrs = {"myattr": attr.string(default="default")}
  )

def _foo_test_impl(env):
  env.expect.that_str(...).equals(...)

def my_test_suite(name):
  test_suite(
    name = name,
    custom_tests = [
      _foo_test,
    ]
  )
```

Note that a custom test is not required to define custom attributes; the above is just an example. If you want to define custom attributes for every test in a suite, the `test_kwargs` argument of `test_suite` can be used to pass additional arguments to all tests in the suite.

3.7 API Reference

3.7.1 ActionSubject

ActionSubject.new

ActionSubject.new(*action*, *meta*)

Creates an "ActionSubject" struct.

Method: ActionSubject.new

Example usage:

```
expect(env).that_action(action).not_contains_arg("foo")
```

PARAMETERS ¶

action¶

(`Action`) value to check against.

meta¶

(*`ExpectMeta`*) of call chain information.

RETURNS ¶

`ActionSubject` object.

ActionSubject.parse_flags

ActionSubject.parse_flags(*argv*)

PARAMETERS ¶

argv¶

undocumented

ActionSubject.argv

ActionSubject.argv()

Returns a CollectionSubject for the action's argv.

Method: ActionSubject.argv

RETURNS ¶

CollectionSubject object.

ActionSubject.contains_at_least_args

ActionSubject.contains_at_least_args(*args*)

Assert that an action contains at least the provided args.

Method: ActionSubject.contains_at_least_args

Example usage: expect(env).that_action(action).contains_at_least_args(["foo", "bar"]).

PARAMETERS ¶

args¶

(list of str) all the args must be in the argv exactly as provided. Multiplicity is respected.

RETURNS ¶

Ordered (see _ordered_incorrectly_new).

ActionSubject.not_contains_arg

ActionSubject.not_contains_arg(*arg*)

Assert that an action does not contain an arg.

Example usage: expect(env).that_action(action).not_contains_arg("should-not-exist")

PARAMETERS ¶

arg¶

(str) the arg that cannot be present in the argv.

ActionSubject.substitutions

ActionSubject.substitutions()

Creates a `DictSubject` to assert on the substitutions dict.

Method: ActionSubject.substitutions.

RETURNS ¶
`DictSubject` struct.

ActionSubject.has_flags_specified

ActionSubject.has_flags_specified(*flags*)

Assert that an action has the given flags present (but ignore any value).

Method: ActionSubject.has_flags_specified

This parses the argv, assuming the typical formats (`--flag=value`, `--flag value`, and `--flag`). Any of the formats will be matched.

Example usage, given `argv = ["--a", "--b=1", "--c", "2"]`: expect(env).that_action(action).has_flags_specified(["–a", "–b", "–c"])

PARAMETERS ¶

flags¶

(`list` of `str`) The flags to check for. Include the leading "–". Multiplicity is respected. A flag is considered present if any of these forms are detected: `--flag=value`, `--flag value`, or a lone `--flag`.

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

ActionSubject.mnemonic

ActionSubject.mnemonic()

Returns a `StrSubject` for the action's mnemonic.

Method: ActionSubject.mnemonic

RETURNS ¶
StrSubject object.

ActionSubject.inputs

ActionSubject.inputs()

Returns a DepsetFileSubject for the action's inputs.

Method: ActionSubject.inputs

RETURNS ¶
`DepsetFileSubject` of the action's inputs.

ActionSubject.contains_flag_values

ActionSubject.contains_flag_values(*flag_values*)

Assert that an action's argv has the given ("–flag", "value") entries.

Method: ActionSubject.contains_flag_values

This parses the argv, assuming the typical formats (`--flag=value`, `--flag value`, and `--flag`). Note, however, that for the `--flag value` and `--flag` forms, the parsing can't know how many args, if any, a flag actually consumes, so it simply takes the first following arg, if any, as the matching value.

NOTE: This function can give misleading results checking flags that don't consume any args (e.g. boolean flags). Use `has_flags_specified()` to test for such flags. Such cases will either show the subsequent arg as the value, or None if the flag was the last arg in argv.

Example usage, given `argv = ["--b=1", "--c", "2"]`: expect(env).that_action(action).contains_flag_values([("–b", "1"), ("–c", "2")])

PARAMETERS ¶

flag_values¶

(`list` of (`str` name, `str`) tuples) Include the leading "–" in the flag name. Order and duplicates aren't checked. Flags without a value found use `None` as their value.

ActionSubject.contains_none_of_flag_values

ActionSubject.contains_none_of_flag_values(*flag_values*)

Assert that an action's argv has none of the given ("–flag", "value") entries.

Method: ActionSubject.contains_none_of_flag_values

This parses the argv, assuming the typical formats (`--flag=value`, `--flag value`, and `--flag`). Note, however, that for the `--flag value` and `--flag` forms, the parsing can't know how many args, if any, a flag actually consumes, so it simply takes the first following arg, if any, as the matching value.

NOTE: This function can give misleading results checking flags that don't consume any args (e.g. boolean flags). Use `has_flags_specified()` to test for such flags.

PARAMETERS ¶

flag_values¶

(`list` of (`str` name, `str` value) tuples) Include the leading "–" in the flag name. Order and duplicates aren't checked.

ActionSubject.contains_at_least_inputs

ActionSubject.contains_at_least_inputs(*inputs*)

Assert the action's inputs contains at least all of `inputs`.

Method: ActionSubject.contains_at_least_inputs

Example usage: expect(env).that_action(action).contains_at_least_inputs([])

PARAMETERS ¶

inputs¶

(collection of `File`) All must be present. Multiplicity is respected.

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

ActionSubject.content

ActionSubject.content()

Returns a `StrSubject` for `Action.content`.

Method: ActionSubject.content

RETURNS ¶
StrSubject object.

ActionSubject.env

ActionSubject.env()

Returns a `DictSubject` for `Action.env`.

Method: ActionSubject.env

3.7.2 Analysis test

Support for testing analysis phase logic, such as rules.

analysis_test

analysis_test(*name*, *target*=None, *targets*=None, *impl*, *expect_failure*=False, *attrs*={}, *attr_values*={}, *fragments*=[], *config_settings*={}, *extra_target_under_test_aspects*=[], *collect_actions_recursively*=False, *provider_subject_factories*=[])

Creates an analysis test from its implementation function.

An analysis test verifies the behavior of a "real" rule target by examining and asserting on the providers given by the real target.

Each analysis test is defined in an implementation function. This function handles the boilerplate to create and return a test target and captures the implementation function's name so that it can be printed in test feedback.

An example of an analysis test:

```
def basic_test(name):
    my_rule(name = name + "_subject", ...)

    analysistest(name = name, target = name + "_subject", impl = _your_test)

def _your_test(env, target, actions):
    env.assert_that(target).runfiles().contains_at_least("foo.txt")
    env.assert_that(find_action(actions, generating="foo.txt")).argv().contains("--a")
```

PARAMETERS ¶

name¶

Name of the target. It should be a Starlark identifier, matching pattern '[A-Za-z_][A-Za-z0-9_]*'.

target

(*default* `None`) The value for the singular target attribute under test. Mutually exclusive with the `targets` arg. The type of value passed determines the type of attribute created:

```
* A list creates an `attr.label_list`
* A dict creates an `attr.label_keyed_string_dict`
* Other values (string and Label) create an `attr.label`.
```

When set, the `impl` function is passed the value for the attribute under test, e.g. passing a list here will pass a list to the `impl` function. These targets all have the target under test config applied.

targets

(*default* `None`) dict of attribute names and their values to test. Mutually exclusive with `target`. Each key must be a valid attribute name and Starlark identifier. When set, the `impl` function is passed a struct of the targets under test, where each attribute corresponds to this dict's keys. The attributes have the target under test config applied and can be customized using `attrs`.

impl

The implementation function of the analysis test.

expect_failure

(*default* `False`) If true, the analysis test will expect the target to fail. Assertions can be made on the underlying failure using truth.expect_failure

attrs

(*default* `{}`) An optional dictionary to supplement the attrs passed to the unit test's `rule()` constructor. Each value can be one of two types:

1. An attribute object, e.g. from `attr.string()`.
2. A dict that describes how to create the attribute; such objects have the target under test settings from other args applied. The dict supports the following keys:
   - `@attr`: A function to create an attribute, e.g. `attr.label`. If unset, `attr.label` is used.
   - `@config_settings`: A dict of config settings; see the `config_settings` argument. These are merged with the `config_settings` arg, with these having precendence.
   - `aspects`: additional aspects to apply in addition to the regular target under test aspects.
   - `cfg`: Not supported; replaced with the target-under-test transition.
   - All other keys are treated as kwargs for the `@attr` function.

attr_values

(*default* `{}`) An optional dictionary of kwargs to pass onto the analysis test target itself (e.g. common attributes like `tags`, `target_compatible_with`, or attributes from `attrs`). Note that these are for the analysis test target itself, not the target under test.

fragments

(*default* `[]`) An optional list of fragment names that can be used to give rules access to language-specific parts of configuration.

config_settings

(*default* `{}`) A dictionary of configuration settings to change for the target under test and its dependencies. This may be used to essentially change 'build flags' for the target under test, and may thus be utilized to test multiple targets with different flags in a single build. NOTE: When values that

are labels (e.g. for the –platforms flag), it's suggested to always explicitly call `Label()` on the value before passing it in. This ensures the label is resolved in your repository's context, not rule_testing's.

extra_target_under_test_aspects¶

(*default `[]`*) An optional list of aspects to apply to the target_under_test in addition to those set up by default for the test harness itself.

collect_actions_recursively¶

(*default `False`*) If true, runs testing_aspect over all attributes, otherwise it is only applied to the target under test.

provider_subject_factories¶

(*default `[]`*) Optional list of ProviderSubjectFactory structs, these are additional provider factories on top of built in ones. A ProviderSubjectFactory is a struct with the following fields:

- type: A provider object, e.g. the callable FooInfo object
- name: A human-friendly name of the provider (eg. "FooInfo")
- factory: A callable to convert an instance of the provider to a subject; see TargetSubject.provider()'s factory arg for the signature.

RETURNS ¶
(None)

test_suite

test_suite(*kwargs*)

This is an alias to lib/test_suite.bzl#test_suite.

PARAMETERS ¶

kwargs¶

Args passed through to test_suite

3.7.3 BoolSubject

BoolSubject.new

BoolSubject.new(*value*, *meta*)

Creates a "BoolSubject" struct.

Method: BoolSubject.new

PARAMETERS ¶

value¶

(`bool`) the value to assert against.

meta¶

(*`ExpectMeta`*) the metadata about the call chain.

RETURNS ¶
A *`BoolSubject`*.

BoolSubject.equals

BoolSubject.equals(*expected*)

Assert that the bool is equal to `expected`.

Method: BoolSubject.equals

PARAMETERS ¶

expected¶

(`bool`) the expected value.

BoolSubject.not_equals

BoolSubject.not_equals(*unexpected*)

Assert that the bool is not equal to `unexpected`.

Method: BoolSubject.not_equals

PARAMETERS ¶

unexpected¶

(`bool`) the value actual cannot equal.

3.7.4 CollectionSubject

CollectionSubject.contains

CollectionSubject.contains(*expected*)

Asserts that `expected` is within the collection.

Method: CollectionSubject.contains

PARAMETERS ¶

expected¶

(`str`) the value that must be present.

CollectionSubject.contains_at_least

CollectionSubject.contains_at_least(*expect_contains*)

Assert that the collection is a subset of the given predicates.

Method: CollectionSubject.contains_at_least

The collection must contain all the values. It can contain extra elements. The multiplicity of values is respected. Checking that the relative order of matches is the same as the passed-in expected values order can done by calling `in_order()`.

PARAMETERS ¶

expect_contains¶

(`list`) values that must be in the collection.

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

CollectionSubject.contains_at_least_predicates

CollectionSubject.contains_at_least_predicates(*matchers*)

Assert that the collection is a subset of the given predicates.

Method: CollectionSubject.contains_at_least_predicates

The collection must match all the predicates. It can contain extra elements. The multiplicity of matchers is respected. Checking that the relative order of matches is the same as the passed-in matchers order can done by calling `in_order()`.

PARAMETERS ¶

matchers¶

(`list` of [`Matcher`]) (see `matchers` struct).

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

CollectionSubject.contains_exactly

CollectionSubject.contains_exactly(*expected*)

Check that a collection contains exactly the given elements.

Method: CollectionSubject.contains_exactly

- Multiplicity is respected.
- The collection must contain all the values, no more or less.
- Checking that the order of matches is the same as the passed-in matchers order can be done by call `in_order()`.

The collection must contain all the values and no more. Multiplicity of values is respected. Checking that the order of matches is the same as the passed-in matchers order can done by calling `in_order()`.

PARAMETERS ¶

expected¶

(`list`) values that must exist.

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

CollectionSubject.contains_exactly_predicates

CollectionSubject.contains_exactly_predicates(*expected*)

Check that the values correspond 1:1 to the predicates.

Method: CollectionSubject.contains_exactly_predicates

- There must be a 1:1 correspondence between the container values and the predicates.
- Multiplicity is respected (i.e., if the same predicate occurs twice, then two distinct elements must match).
- Matching occurs in first-seen order. That is, a predicate will "consume" the first value in `actual_container` it matches.

- The collection must match all the predicates, no more or less.
- Checking that the order of matches is the same as the passed-in matchers order can be done by call `in_order()`.

Note that confusing results may occur if predicates with overlapping match conditions are used. For example, given: actual=[“a”, “ab”, “abc”], predicates=[, ,]

Then the result will be they aren't equal: the first two predicates consume “a” and “ab”, leaving only “abc” for the predicate to match against, which fails.

PARAMETERS ¶

expected¶

(`list` of [`Matcher`]) that must match.

RETURNS ¶

Ordered (see `_ordered_incorrectly_new`).

CollectionSubject.contains_none_of

CollectionSubject.contains_none_of(*values*)

Asserts the collection contains none of `values`.

Method: CollectionSubject.contains_none_of

PARAMETERS ¶

values¶

([`collection`]) values of which none of are allowed to exist.

CollectionSubject.contains_predicate

CollectionSubject.contains_predicate(*matcher*)

Asserts that `matcher` matches at least one value.

Method: CollectionSubject.contains_predicate

PARAMETERS ¶

matcher¶

([`Matcher`]) (see `matchers` struct).

CollectionSubject.has_size

CollectionSubject.has_size(*expected*)

Asserts that `expected` is the size of the collection.

Method: CollectionSubject.has_size

PARAMETERS ¶

expected¶

([`int`]) the expected size of the collection.

CollectionSubject.new

CollectionSubject.new(*values*, *meta*, *container_name*="values", *sortable*=True, *element_plural_name*="elements")

Creates a "CollectionSubject" struct.

Method: CollectionSubject.new

Public Attributes:

- `actual`: The wrapped collection.

PARAMETERS ¶

values¶

([`collection`]) the values to assert against.

meta¶

(*ExpectMeta*) the metadata about the call chain.

container_name¶

(*default "values"*) (`str`) conceptual name of the container.

sortable¶

(*default True*) (`bool`) True if output should be sorted for display, False if not.

element_plural_name¶

(*default "elements"*) (`str`) the plural word for the values in the container.

RETURNS ¶

CollectionSubject.

CollectionSubject.not_contains_predicate

CollectionSubject.not_contains_predicate(*matcher*)

Asserts that `matcher` matches no values in the collection.

Method: CollectionSubject.not_contains_predicate

PARAMETERS ¶

matcher¶

[`Matcher`] object (see `matchers` struct).

CollectionSubject.offset

CollectionSubject.offset(*offset*, *factory*)

Fetches an element from the collection as a subject.

PARAMETERS ¶

offset¶

([`int`]) the offset to fetch

factory¶

([callable]). The factory function to use to create the subject for the offset's value. It must have the following signature: `def factory(value, *, meta)`.

RETURNS ¶

Object created by `factory`.

CollectionSubject.transform

CollectionSubject.transform(*desc*=None, *map_each*=None, *loop*=None, *filter*=None)

Transforms a collections's value and returns another CollectionSubject.

This is equivalent to applying a list comprehension over the collection values, but takes care of propagating context information and wrapping the value in a `CollectionSubject`.

`transform(map_each=M, loop=L, filter=F)` is equivalent to `[M(v) for v in L(collection) if F(v)]`.

PARAMETERS ¶

desc¶

(*default* `None`) (optional `str`) a human-friendly description of the transform for use in error messages. Required when a description can't be inferred from the other args. The description can be inferred if the filter arg is a named function (non-lambda) or Matcher object.

map_each¶

(*default* `None`) (optional [`callable`]) function to transform an element in the collection. It takes one positional arg, the loop's current iteration value, and its return value will be the element's new value. If not specified, the values from the loop iteration are returned unchanged.

loop¶

(*default* `None`) (optional [`callable`]) function to produce values from the original collection and whose values are iterated over. It takes one positional arg, which is the original collection. If not specified, the original collection values are iterated over.

filter¶

(*default* `None`) (optional [`callable`]) function that decides what values are passed onto `map_each` for inclusion in the final result. It takes one positional arg, the value to match (which is the current iteration value before `map_each` is applied), and returns a bool (True if the value should be included in the result, False if it should be skipped).

RETURNS ¶

`CollectionSubject` of the transformed values.

3.7.5 DefaultInfoSubject

DefaultInfoSubject.new

DefaultInfoSubject.new(*info*, *meta*)

Creates a `DefaultInfoSubject`

PARAMETERS ¶

info¶

([`DefaultInfo`]) the DefaultInfo object to wrap.

meta

(*ExpectMeta*) call chain information.

RETURNS
[DefaultInfoSubject] object.

DefaultInfoSubject.runfiles

DefaultInfoSubject.runfiles()

Creates a subject for the default runfiles.

RETURNS
RunfilesSubject object

DefaultInfoSubject.data_runfiles

DefaultInfoSubject.data_runfiles()

Creates a subject for the data runfiles.

RETURNS
RunfilesSubject object

DefaultInfoSubject.default_outputs

DefaultInfoSubject.default_outputs()

Creates a subject for the default outputs.

RETURNS
DepsetFileSubject object.

DefaultInfoSubject.executable

DefaultInfoSubject.executable()

Creates a subject for the executable file.

RETURNS
FileSubject object.

DefaultInfoSubject.runfiles_manifest

DefaultInfoSubject.runfiles_manifest()

Creates a subject for the runfiles manifest.

RETURNS
FileSubject object.

3.7.6 DepsetFileSubject

DepsetFileSubject.new

DepsetFileSubject.new(*files*, *meta*, *container_name*="depset", *element_plural_name*="files")

Creates a DepsetFileSubject asserting on `files`.

Method: DepsetFileSubject.new

PARAMETERS ¶

files¶

(`depset` of `File`) the values to assert on.

meta¶

(*ExpectMeta*) of call chain information.

container_name¶

(*default "depset"*) (`str`) conceptual name of the container.

element_plural_name¶

(*default "files"*) (`str`) the plural word for the values in the container.

RETURNS ¶

DepsetFileSubject object.

DepsetFileSubject.contains

DepsetFileSubject.contains(*expected*)

Asserts that the depset of files contains the provided path/file.

Method: DepsetFileSubject.contains

PARAMETERS ¶

expected¶

(`str` | `File`) If a string path is provided, it is compared to the short path of the files and are formatted using [`ExpectMeta.format_str`] and its current contextual keywords. Note that, when using `File` objects, two files' configurations must be the same for them to be considered equal.

DepsetFileSubject.contains_at_least

DepsetFileSubject.contains_at_least(*expected*)

Asserts that the depset of files contains at least the provided paths.

Method: DepsetFileSubject.contains_at_least

PARAMETERS ¶

expected¶

([`collection`] of `str` | collection of `File`) multiplicity is respected. If string paths are provided, they are compared to the short path of the files and are formatted using `ExpectMeta.format_str` and its current contextual keywords. Note that, when using `File` objects, two files' configurations must be the same for them to be considered equal.

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

DepsetFileSubject.contains_any_in

DepsetFileSubject.contains_any_in(*expected*)

Asserts that any of the values in `expected` exist.

Method: DepsetFileSubject.contains_any_in

PARAMETERS ¶

expected¶

([`collection`] of `str` paths | [`collection`] of `File`) at least one of the values must exist. Note that, when using `File` objects, two files' configurations must be the same for them to be considered equal. When string paths are provided, they are compared to `File.short_path`.

DepsetFileSubject.contains_at_least_predicates

DepsetFileSubject.contains_at_least_predicates(*matchers*)

Assert that the depset is a subset of the given predicates.

Method: DepsetFileSubject.contains_at_least_predicates

The depset must match all the predicates. It can contain extra elements. The multiplicity of matchers is respected. Checking that the relative order of matches is the same as the passed-in matchers order can done by calling `in_order()`.

PARAMETERS ¶

matchers¶

(`list` of [`Matcher`]) (see `matchers` struct) that accept `File` objects.

RETURNS ¶
Ordered (see `_ordered_incorrectly_new`).

DepsetFileSubject.contains_predicate

DepsetFileSubject.contains_predicate(*matcher*)

Asserts that `matcher` matches at least one value.

Method: DepsetFileSubject.contains_predicate

PARAMETERS ¶

matcher¶

[`Matcher`] (see `matching` struct) that accepts `File` objects.

DepsetFileSubject.contains_exactly

DepsetFileSubject.contains_exactly(*paths*)

Asserts the depset of files contains exactly the given paths.

Method: DepsetFileSubject.contains_exactly

PARAMETERS ¶

paths¶

([collection] of str) the paths that must exist. These are compared to the short_path values of the files in the depset. All the paths, and no more, must exist.

DepsetFileSubject.not_contains

DepsetFileSubject.not_contains(*short_path*)

Asserts that short_path is not in the depset.

Method: DepsetFileSubject.not_contains_predicate

PARAMETERS ¶

short_path¶

(str) the short path that should not be present.

DepsetFileSubject.not_contains_predicate

DepsetFileSubject.not_contains_predicate(*matcher*)

Asserts that nothing in the depset matches matcher.

Method: DepsetFileSubject.not_contains_predicate

PARAMETERS ¶

matcher¶

([Matcher]) that must match. It operates on File objects.

3.7.7 DictSubject

DictSubject.new

DictSubject.new(*actual*, *meta*, *container_name*="dict", *key_plural_name*="keys")

Creates a new DictSubject.

Method: DictSubject.new

PARAMETERS ¶

actual¶

(dict) the dict to assert against.

meta¶

(*ExpectMeta*) of call chain information.

container_name¶

(*default* `"dict"`) (`str`) conceptual name of the dict.

key_plural_name¶

(*default* `"keys"`) (`str`) the plural word for the keys of the dict.

RETURNS ¶
New `DictSubject` struct.

DictSubject.contains_at_least

DictSubject.contains_at_least(*at_least*)

Assert the dict has at least the entries from `at_least`.

Method: DictSubject.contains_at_least

PARAMETERS ¶

at_least¶

(`dict`) the subset of keys/values that must exist. Extra keys are allowed. Order is not checked.

DictSubject.contains_exactly

DictSubject.contains_exactly(*expected*)

Assert the dict has exactly the provided values.

Method: DictSubject.contains_exactly

PARAMETERS ¶

expected¶

(`dict`) the values that must exist. Missing values or extra values are not allowed. Order is not checked.

DictSubject.contains_none_of

DictSubject.contains_none_of(*none_of*)

Assert the dict contains none of `none_of` keys/values.

Method: DictSubject.contains_none_of

PARAMETERS ¶

none_of¶

(`dict`) the keys/values that must not exist. Order is not checked.

DictSubject.keys

DictSubject.keys()

Returns a `CollectionSubject` for the dict's keys.

Method: DictSubject.keys

RETURNS ¶
CollectionSubject of the keys.

3.7.8 ExecutionInfoSubject

ExecutionInfoSubject.new

ExecutionInfoSubject.new(*info*, *meta*)

Create a new `ExecutionInfoSubject`

Method: ExecutionInfoSubject.new

PARAMETERS ¶

info¶

([`testing.ExecutionInfo`]) provider instance.

meta¶

(*ExpectMeta*) of call chain information.

RETURNS ¶
`ExecutionInfoSubject` struct.

ExecutionInfoSubject.requirements

ExecutionInfoSubject.requirements()

Create a `DictSubject` for the requirements values.

Method: ExecutionInfoSubject.requirements

RETURNS ¶
`DictSubject` of the requirements.

ExecutionInfoSubject.exec_group

ExecutionInfoSubject.exec_group()

Create a `StrSubject` for the `exec_group` value.

Method: ExecutionInfoSubject.exec_group

RETURNS ¶
A *StrSubject* for the exec group.

3.7.9 Expect

Expect.new

Expect.new(*env*, *meta*)

Creates a new Expect object.

Internal; only other `Expect` methods should be calling this.

PARAMETERS ¶

env¶

unittest env struct or some approximation.

meta¶

(*ExpectMeta*) metadata about call chain and state.

RETURNS ¶

Expect object

Expect.new_from_env

Expect.new_from_env(*env*)

Wrapper around `env`.

This is the entry point to the Truth-style assertions. Example usage: expect = expect(env) expect.that_action(action).contains_at_least_args(...)

The passed in `env` object allows optional attributes to be set to customize behavior. Usually this is helpful for testing. See `_fake_env()` in truth_tests.bzl for examples.

- `fail`: callable that takes a failure message. If present, it will be called instead of the regular `Expect.add_failure` logic.
- `get_provider`: callable that takes 2 positional args (target and provider) and returns the found provider or fails.
- `has_provider`: callable that takes 2 positional args (a `Target` and a [`provider`]) and returns `bool` (`True` if present, `False` otherwise) or fails.

PARAMETERS ¶

env¶

unittest env struct, or some approximation. There are several attributes that override regular behavior; see above doc.

RETURNS ¶

Expect object

Expect.that_action

Expect.that_action(*action*)

Creates a subject for asserting Actions.

PARAMETERS ¶

action¶

(`Action`) the action to check.

RETURNS ¶

ActionSubject object.

Expect.that_bool

Expect.that_bool(*value*, *expr*="boolean")

Creates a subject for asserting a boolean.

PARAMETERS ¶

value¶

(`bool`) the bool to check.

expr¶

(*default* `"boolean"`) (`str`) the starting "value of" expression to report in errors.

RETURNS ¶

BoolSubject object.

Expect.that_collection

Expect.that_collection(*collection*, *expr*="collection", *kwargs*)

Creates a subject for asserting collections.

PARAMETERS ¶

collection¶

The collection (list or depset) to assert.

expr¶

(*default* `"collection"`) (`str`) the starting "value of" expression to report in errors.

kwargs¶

Additional kwargs to pass onto CollectionSubject.new

RETURNS ¶

CollectionSubject object.

Expect.that_depset_of_files

Expect.that_depset_of_files(*depset_files*)

Creates a subject for asserting a depset of files.

Method: Expect.that_depset_of_files

PARAMETERS ¶

depset_files¶

(`depset` of `File`) the values to assert on.

RETURNS ¶

DepsetFileSubject object.

Expect.that_dict

Expect.that_dict(*mapping*, *meta*=None)

Creates a subject for asserting a dict.

Method: Expect.that_dict

PARAMETERS ¶

mapping¶

(`dict`) the values to assert on

meta¶

(*default* `None`) (*ExpectMeta*) optional custom call chain information to use instead

RETURNS ¶

DictSubject object.

Expect.that_file

Expect.that_file(*file*, *meta*=None)

Creates a subject for asserting a file.

Method: Expect.that_file

PARAMETERS ¶

file¶

(`File`) the value to assert.

meta¶

(*default* `None`) (*ExpectMeta*) optional custom call chain information to use instead

RETURNS ¶

FileSubject object.

Expect.that_int

Expect.that_int(*value*, *expr*="integer")

Creates a subject for asserting an `int`.

Method: Expect.that_int

PARAMETERS ¶

value¶

([`int`]) the value to check against.

expr¶

(*default `"integer"`*) (`str`) the starting "value of" expression to report in errors.

RETURNS ¶

`IntSubject` object.

Expect.that_str

Expect.that_str(*value*)

Creates a subject for asserting a `str`.

PARAMETERS ¶

value¶

(`str`) the value to check against.

RETURNS ¶

`StrSubject` object.

Expect.that_struct

Expect.that_struct(*value*, *attrs*, *expr*="struct")

Creates a subject for asserting a `struct`.

PARAMETERS ¶

value¶

(`struct`) the value to check against.

attrs¶

(`dict` of `str` to [`callable`]) the functions to convert attributes to subjects. The keys are attribute names that must exist on `actual`. The values are functions with the signature `def factory(value, *, meta)`, where `value` is the actual attribute value of the struct, and `meta` is an *`ExpectMeta`* object.

expr¶

(*default `"struct"`*) (`str`) The starting "value of" expression to report in errors.

RETURNS ¶

`StructSubject` object.

Expect.that_target

Expect.that_target(*target*)

Creates a subject for asserting a `Target`.

This adds the following parameters to `ExpectMeta.format_str`: {package}: The target's package, e.g. "foo/bar" from "//foo/bar:baz" {name}: The target's base name, e.g., "baz" from "//foo/bar:baz"

PARAMETERS ¶

target¶

(`Target`) subject target to check against.

RETURNS ¶

TargetSubject object.

Expect.that_value

Expect.that_value(*value*, *factory*, *expr*="value")

Creates a subject for asserting an arbitrary value of a custom type.

PARAMETERS ¶

value¶

(`struct`) the value to check against.

factory¶

A subject factory (a function that takes value and meta). Eg. subjects.collection

expr¶

(*default `"value"`*) (`str`) The starting "value of" expression to report in errors.

RETURNS ¶

A subject corresponding to the type returned by the factory.

Expect.where

Expect.where(*details*)

Add additional information about the assertion.

This is useful for attaching information that isn't part of the call chain or some reason. Example usage:

```
expect(env).where(platform=ctx.attr.platform).that_str(...)
```

Would include "platform: {ctx.attr.platform}" in failure messages.

PARAMETERS ¶

details¶

(`dict` of `str` to value) Each named arg is added to the metadata details with the provided string, which is printed as part of displaying any failures.

RETURNS ¶

Expect object with separate metadata derived from the original self.

3.7.10 ExpectMeta

ExpectMeta object implementation.

ExpectMeta.new

ExpectMeta.new(*env*, *exprs*=[], *details*=[], *format_str_kwargs*=None)

Creates a new "ExpectMeta" struct".

Method: ExpectMeta.new

ExpectMeta objects are internal helpers for the Expect object and Subject objects. They are used for Subjects to store and communicate state through a series of call chains and asserts.

This constructor should only be directly called by `Expect` objects. When a parent Subject is creating a child-Subject, then [`derive()`] should be used.

Env objects

The `env` object basically provides a way to interact with things outside of the truth assertions framework. This allows easier testing of the framework itself and decouples it from a particular test framework (which makes it usable by by rules_testing's analysis_test and skylib's analysistest)

The `env` object requires the following attribute:

- ctx: The test's ctx.

The `env` object allows the following attributes to customize behavior:

- fail: A callable that accepts a single string, which is the failure message. Its return value is ignored. This is called when an assertion fails. It's generally expected that it records a failure instead of immediately failing.
- has_provider: (callable) it accepts two positional args, target and provider and returns `bool`. This is used to implement `Provider in target` operations.
- get_provider: (callable) it accepts two positional args, target and provider and returns the provider value. This is used to implement `target[Provider]`.

PARAMETERS ¶

env¶

unittest env struct or some approximation.

exprs¶

(*default `[]`*) (`list` of `str`) the expression strings of the call chain for the subject.

details¶

(*default `[]`*) (`list` of `str`) additional details to print on error. These are usually informative details of the objects under test.

format_str_kwargs¶

(*default `None`*) optional dict of format() kwargs. These kwargs are propagated through `derive()` calls and used when `ExpectMeta.format_str()` is called.

RETURNS ¶

`ExpectMeta` object.

ExpectMeta.derive

ExpectMeta.derive(*expr*=None, *details*=None, *format_str_kwargs*={})

Create a derivation of the current meta object for a child-Subject.

Method: ExpectMeta.derive

When a Subject needs to create a child-Subject, it derives a new meta object to pass to the child. This separates the parent's state from the child's state and allows any failures generated by the child to include the context of the parent creator.

Example usage:

```
def _foo_subject_action_named(self, name):
    meta = self.meta.derive("action_named({})".format(name),
                            "action: {}".format(...))
    return ActionSubject(..., meta)
def _foo_subject_name(self):
    # No extra detail to include)
    meta self.meta.derive("name()", None)
```

PARAMETERS ¶

expr¶

(*default* `None`) (`str`) human-friendly description of the call chain expression. e.g., if `foo_subject.bar_named("baz")` returns a child-subject, then "bar_named("bar")" would be the expression.

details¶

(*default* `None`) (optional `list` of `str`) human-friendly descriptions of additional detail to include in errors. This is usually additional information the child Subject wouldn't include itself. e.g. if `foo.first_action_argv().contains(1)`, returned a ListSubject, then including "first action: Action FooCompile" helps add context to the error message. If there is no additional detail to include, pass None.

format_str_kwargs¶

(*default* `{}`) (`dict` of format()-kwargs) additional kwargs to make available to *`format_str`* calls.

RETURNS ¶

`ExpectMeta` object.

ExpectMeta.format_str

ExpectMeta.format_str(*template*)

Interpolate contextual keywords into a string.

This uses the normal `format()` style (i.e. using `{}`). Keywords refer to parts of the call chain.

The particular keywords supported depend on the call chain. The following are always present: {workspace}: The name of the workspace, e.g. "rules_proto". {test_name}: The base name of the current test.

PARAMETERS ¶

template¶

(`str`) the format template string to use.

RETURNS ¶
str; the template with parameters replaced.

ExpectMeta.get_provider

ExpectMeta.get_provider(*target*, *provider*)

Get a provider from a target.

This is equivalent to `target[provider]`; the extra level of indirection is to aid testing.

PARAMETERS ¶

target¶

(Target) the target to get the provider from.

provider¶

The provider type to get.

RETURNS ¶
The found provider, or fails if not present.

ExpectMeta.has_provider

ExpectMeta.has_provider(*target*, *provider*)

Tells if a target has a provider.

This is equivalent to `provider in target`; the extra level of indirection is to aid testing.

PARAMETERS ¶

target¶

(Target) the target to check for the provider.

provider¶

the provider type to check for.

RETURNS ¶
True if the target has the provider, False if not.

ExpectMeta.add_failure

ExpectMeta.add_failure(*problem*, *actual*)

Adds a failure with context.

Method: ExpectMeta.add_failure

Adds the given error message. Context from the subject and prior call chains is automatically added.

PARAMETERS ¶

problem¶

(str) a string describing the expected value or problem detected, and the expected values that weren't satisfied. A colon should be used to separate the description from the values. The description should be brief and include the word "expected", e.g. "expected: foo", or "expected values missing: ",

the key point being the reader can easily take the values shown and look for it in the actual values displayed below it.

actual

(str) a string describing the values observed. A colon should be used to separate the description from the observed values. The description should be brief and include the word "actual", e.g., "actual: bar". The values should include the actual, observed, values and pertinent information about them.

ExpectMeta.call_fail

ExpectMeta.call_fail(*msg*)

Adds a failure to the test run.

PARAMETERS

msg

(str) the failure message.

3.7.11 FileSubject

FileSubject.new

FileSubject.new(*file*, *meta*)

Creates a FileSubject asserting against the given file.

Method: FileSubject.new

PARAMETERS

file

(File) the file to assert against.

meta

(*ExpectMeta*)

RETURNS

FileSubject object.

FileSubject.equals

FileSubject.equals(*expected*)

Asserts that `expected` references the same file as `self`.

This uses Bazel's notion of File equality, which usually includes the configuration, owning action, internal hash, etc of a `File`. The particulars of comparison depend on the actual Java type implementing the `File` object (some ignore owner, for example).

NOTE: This does not compare file content. Starlark cannot read files.

NOTE: Same files generated by different owners are likely considered not equal to each other. The alternative for this is to assert the `File.path` paths are equal using [`FileSubject.path()`]

Method: FileSubject.equals

PARAMETERS

expected¶

undocumented

FileSubject.path

FileSubject.path()

Returns a `StrSubject` asserting on the files `path` value.

Method: FileSubject.path

RETURNS ¶
`StrSubject` object.

FileSubject.short_path_equals

FileSubject.short_path_equals(*path*)

Asserts the file's short path is equal to the given path.

Method: FileSubject.short_path_equals

PARAMETERS ¶

path¶

(`str`) the value the file's `short_path` must be equal to.

3.7.12 InstrumentedFilesInfoSubject

InstrumentedFilesInfoSubject.new

InstrumentedFilesInfoSubject.new(*info*, *meta*)

Creates a subject to assert on `InstrumentedFilesInfo` providers.

Method: InstrumentedFilesInfoSubject.new

PARAMETERS ¶

info¶

([`InstrumentedFilesInfo`]) provider instance.

meta¶

(*ExpectMeta*) the meta data about the call chain.

RETURNS ¶
An `InstrumentedFilesInfoSubject` struct.

InstrumentedFilesInfoSubject.instrumented_files

InstrumentedFilesInfoSubject.instrumented_files()

Returns a `DesetFileSubject` of the instrumented files.

Method: InstrumentedFilesInfoSubject.instrumented_files

InstrumentedFilesInfoSubject.metadata_files

InstrumentedFilesInfoSubject.metadata_files()

Returns a `DesetFileSubject` of the metadata files.

Method: InstrumentedFilesInfoSubject.metadata_files

3.7.13 IntSubject

IntSubject.new

IntSubject.new(*value*, *meta*)

Create an "IntSubject" struct.

Method: IntSubject.new

PARAMETERS ¶

value¶

(optional [`int`]) the value to perform asserts against may be None.

meta¶

(*ExpectMeta*) the meta data about the call chain.

RETURNS ¶

IntSubject.

IntSubject.equals

IntSubject.equals(*other*)

Assert that the subject is equal to the given value.

Method: IntSubject.equals

PARAMETERS ¶

other¶

([`int`]) value the subject must be equal to.

IntSubject.is_greater_than

IntSubject.is_greater_than(*other*)

Asserts that the subject is greater than the given value.

Method: IntSubject.is_greater_than

PARAMETERS ¶

other¶

([int]) value the subject must be greater than.

IntSubject.not_equals

IntSubject.not_equals(*unexpected*)

Assert that the int is not equal to `unexpected`.

Method: IntSubject.not_equals

PARAMETERS ¶

unexpected¶

([int]) the value actual cannot equal.

3.7.14 LabelSubject

LabelSubject.new

LabelSubject.new(*label*, *meta*)

Creates a new `LabelSubject` for asserting `Label` objects.

Method: LabelSubject.new

PARAMETERS ¶

label¶

(`Label`) the label to check against.

meta¶

(*`ExpectMeta`*) the metadata about the call chain.

RETURNS ¶

`LabelSubject`.

LabelSubject.equals

LabelSubject.equals(*other*)

Asserts the label is equal to `other`.

Method: LabelSubject.equals

PARAMETERS ¶

other¶

(`Label` | `str`) the expected value. If a `str` is passed, it will be converted to a `Label` using the `Label` function.

LabelSubject.is_in

LabelSubject.is_in(*any_of*)

Asserts that the label is any of the provided values.

PARAMETERS ¶

any_of¶

([`collection`] of (`Label` | `str`)) If strings are provided, they must be parsable by `Label`.

3.7.15 Ordered

IN_ORDER.in_order

IN_ORDER.in_order()

Checks that the values were in order.

OrderedIncorrectly.new

OrderedIncorrectly.new(*format_problem*, *format_actual*, *meta*)

Creates a new `Ordered` object that fails due to incorrectly ordered values.

This creates an *Ordered* object that always fails. If order is correct, use the `_IN_ORDER` constant.

PARAMETERS ¶

format_problem¶

(callable) accepts no args and returns string (the reported problem description).

format_actual¶

(callable) accepts not args and returns string (the reported actual description).

meta¶

(*ExpectMeta*) used to report the failure.

RETURNS ¶

Ordered object.

OrderedIncorrectly.in_order

OrderedIncorrectly.in_order()

Checks that the values were in order.

3.7.16 RunEnvironmentInfoSubject

RunEnvironmentInfoSubject.new

RunEnvironmentInfoSubject.new(*info*, *meta*)

Creates a new `RunEnvironmentInfoSubject`

Method: RunEnvironmentInfoSubject.new

PARAMETERS ¶

info¶

([`RunEnvironmentInfo`]) provider instance.

meta¶

(*ExpectMeta*) of call chain information.

RunEnvironmentInfoSubject.environment

RunEnvironmentInfoSubject.environment()

Creates a `DictSubject` to assert on the environment dict.

Method: RunEnvironmentInfoSubject.environment

RETURNS ¶

DictSubject of the str->str environment map.

RunEnvironmentInfoSubject.inherited_environment

RunEnvironmentInfoSubject.inherited_environment()

Creates a `CollectionSubject` to assert on the inherited_environment list.

Method: RunEnvironmentInfoSubject.inherited_environment

RETURNS ¶

CollectionSubject of `str`; from the [`RunEnvironmentInfo.inherited_environment`] list.

3.7.17 RunfilesSubject

RunfilesSubject.new

RunfilesSubject.new(*runfiles*, *meta*, *kind*=None)

Creates a "RunfilesSubject" struct.

Method: RunfilesSubject.new

PARAMETERS ¶

runfiles¶

([runfiles]) the runfiles to check against.

meta¶

(*ExpectMeta*) the metadata about the call chain.

kind¶

(*default None*) (optional str) what type of runfiles they are, usually "data" or "default". If not known or not applicable, use None.

RETURNS ¶

RunfilesSubject object.

RunfilesSubject.contains

RunfilesSubject.contains(*expected*)

Assert that the runfiles contains the provided path.

Method: RunfilesSubject.contains

PARAMETERS ¶

expected¶

(str) the path to check is present. This will be formatted using ExpectMeta.format_str and its current contextual keywords. Note that paths are runfiles-root relative (i.e. you likely need to include the workspace name.)

RunfilesSubject.contains_at_least

RunfilesSubject.contains_at_least(*paths*)

Assert that the runfiles contains at least all of the provided paths.

Method: RunfilesSubject.contains_at_least

All the paths must exist, but extra paths are allowed. Order is not checked. Multiplicity is respected.

PARAMETERS ¶

paths¶

((collection of str) | [runfiles]) the paths that must exist. If a collection of strings is provided, they will be formatted using [ExpectMeta.format_str], so its template keywords can be directly passed. If a runfiles object is passed, it is converted to a set of path strings.

RunfilesSubject.contains_predicate

RunfilesSubject.contains_predicate(*matcher*)

Asserts that matcher matches at least one value.

Method: RunfilesSubject.contains_predicate

PARAMETERS ¶

matcher¶

callable that takes 1 positional arg (str path) and returns boolean.

RunfilesSubject.contains_exactly

RunfilesSubject.contains_exactly(*paths*)

Asserts that the runfiles contains_exactly the set of paths

Method: RunfilesSubject.contains_exactly

PARAMETERS ¶

paths¶

([collection] of str) the paths to check. These will be formatted using `meta.format_str`, so its template keywords can be directly passed. All the paths must exist in the runfiles exactly as provided, and no extra paths may exist.

RunfilesSubject.contains_none_of

RunfilesSubject.contains_none_of(*paths*, *require_workspace_prefix*=True)

Asserts the runfiles contain none of `paths`.

Method: RunfilesSubject.contains_none_of

PARAMETERS ¶

paths¶

([collection] of str) the paths that should not exist. They should be runfiles root-relative paths (not workspace relative). The value is formatted using `ExpectMeta.format_str` and the current contextual keywords.

require_workspace_prefix¶

(*default `True`*) (bool) True to check that the path includes the workspace prefix. This is to guard against accidentallly passing a workspace relative path, which will (almost) never exist, and cause the test to always pass. Specify False if the file being checked for is *actually* a runfiles-root relative path that isn't under the workspace itself.

RunfilesSubject.not_contains

RunfilesSubject.not_contains(*path*, *require_workspace_prefix*=True)

Assert that the runfiles does not contain the given path.

Method: RunfilesSubject.not_contains

PARAMETERS ¶

path¶

(str) the path that should not exist. It should be a runfiles root-relative path (not workspace relative). The value is formatted using `format_str`, so its template keywords can be directly passed.

require_workspace_prefix¶

(*default `True`*) (bool) True to check that the path includes the workspace prefix. This is to guard against accidentallly passing a workspace relative path, which will (almost) never exist, and cause the test to always pass. Specify False if the file being checked for is *actually* a runfiles-root relative path that isn't under the workspace itself.

RunfilesSubject.not_contains_predicate

RunfilesSubject.not_contains_predicate(*matcher*)

Asserts that none of the runfiles match `matcher`.

Method: RunfilesSubject.not_contains_predicate

PARAMETERS ¶

matcher¶

[`Matcher`] that accepts a string (runfiles root-relative path).

RunfilesSubject.check_workspace_prefix

RunfilesSubject.check_workspace_prefix(*path*)

PARAMETERS ¶

path¶

undocumented

3.7.18 StrSubject

StrSubject.new

StrSubject.new(*actual*, *meta*)

Creates a subject for asserting strings.

Method: StrSubject.new

PARAMETERS ¶

actual¶

(`str`) the string to check against.

meta¶

(*`ExpectMeta`*) of call chain information.

RETURNS ¶

`StrSubject` object.

StrSubject.contains

StrSubject.contains(*substr*)

Assert that the subject contains the substring `substr`.

Method: StrSubject.contains

PARAMETERS ¶

substr¶

(`str`) the substring to check for.

StrSubject.equals

StrSubject.equals(*other*)

Assert that the subject string equals the other string.

Method: StrSubject.equals

PARAMETERS ¶

other¶

(`str`) the expected value it should equal.

StrSubject.not_equals

StrSubject.not_equals(*unexpected*)

Assert that the string is not equal to `unexpected`.

Method: BoolSubject.not_equals

PARAMETERS ¶

unexpected¶

(`str`) the value actual cannot equal.

StrSubject.split

StrSubject.split(*sep*)

Return a `CollectionSubject` for the actual string split by `sep`.

Method: StrSubject.split

PARAMETERS ¶

sep¶

undocumented

3.7.19 StructSubject

A subject for arbitrary structs. This is most useful when wrapping an ad-hoc struct (e.g. a struct specific to a particular function). Such ad-hoc structs are usually just plain data objects, so they don't need special functionality that writing a full custom subject allows. If a struct would benefit from custom accessors or asserts, write a custom subject instead.

This subject is usually used as a helper to a more formally defined subject that knows the shape of the struct it needs to wrap. For example, a `FooInfoSubject` implementation might use it to handle `FooInfo.struct_with_a_couple_fields`.

Note the resulting subject object is not a direct replacement for the struct being wrapped: * Structs wrapped by this subject have the attributes exposed as functions, not as plain attributes. This matches the other subject classes and defers converting an attribute to a subject unless necessary. * The attribute name `actual` is reserved.

Example usages

To use it as part of a custom subject returning a sub-value, construct it using `subjects.struct()` like so:

```
load("@rules_testing//lib:truth.bzl", "subjects")

def _my_subject_foo(self):
    return subjects.struct(
        self.actual.foo,
        meta = self.meta.derive("foo()"),
        attrs = dict(a=subjects.int, b=subjects.str),
    )
```

If you're checking a struct directly in a test, then you can use `Expect.that_struct`. You'll still have to pass the `attrs` arg so it knows how to map the attributes to the matching subject factories.

```
def _foo_test(env):
    actual = env.expect.that_struct(
        struct(a=1, b="x"),
        attrs = dict(a=subjects.int, b=subjects.str)
    )
    actual.a().equals(1)
    actual.b().equals("x")
```

StructSubject.new

StructSubject.new(*actual*, *meta*, *attrs*)

Creates a `StructSubject`, which is a thin wrapper around a `struct`.

PARAMETERS ¶

actual¶

(`struct`) the struct to wrap.

meta¶

(*`ExpectMeta`*) object of call context information.

attrs¶

(`dict` of `str` to [`callable`]) the functions to convert attributes to subjects. The keys are attribute names that must exist on `actual`. The values are functions with the signature `def factory(value, *, meta)`, where `value` is the actual attribute value of the struct, and `meta` is an *`ExpectMeta`* object.

RETURNS ¶

`StructSubject` object, which is a struct with the following shape: * `actual` attribute, the underlying struct that was wrapped. * A callable attribute for each `attrs` entry; it takes no args and returns what the corresponding factory from `attrs` returns.

3.7.20 TargetSubject

`TargetSubject` wraps a `Target` object and provides method for asserting its state.

TargetSubject.new

TargetSubject.new(*target*, *meta*)

Creates a subject for asserting Targets.

Method: TargetSubject.new

Public attributes:

- `actual`: The wrapped `Target` object.

PARAMETERS ¶

target¶

(`Target`) the target to check against.

meta¶

(*`ExpectMeta`*) metadata about the call chain.

RETURNS ¶

`TargetSubject` object

TargetSubject.runfiles

TargetSubject.runfiles()

Creates a subject asserting on the target's default runfiles.

Method: TargetSubject.runfiles

RETURNS ¶

`RunfilesSubject` object.

TargetSubject.tags

TargetSubject.tags()

Gets the target's tags as a `CollectionSubject`

Method: TargetSubject.tags

RETURNS ¶

`CollectionSubject` asserting the target's tags.

TargetSubject.get_attr

TargetSubject.get_attr(*name*)

PARAMETERS ¶

name¶

undocumented

TargetSubject.data_runfiles

TargetSubject.data_runfiles()

Creates a subject asserting on the target's data runfiles.

Method: TargetSubject.data_runfiles

RETURNS ¶

RunfilesSubject object

TargetSubject.default_outputs

TargetSubject.default_outputs()

Creates a subject asserting on the target's default outputs.

Method: TargetSubject.default_outputs

RETURNS ¶

DepsetFileSubject object.

TargetSubject.executable

TargetSubject.executable()

Creates a subject asesrting on the target's executable File.

Method: TargetSubject.executable

RETURNS ¶

FileSubject object.

TargetSubject.failures

TargetSubject.failures()

Creates a subject asserting on the target's failure message strings.

Method: TargetSubject.failures

RETURNS ¶

CollectionSubject of `str`.

TargetSubject.has_provider

TargetSubject.has_provider(*provider*)

Asserts that the target as provider `provider`.

Method: TargetSubject.has_provider

PARAMETERS ¶

provider¶

The provider object to check for.

TargetSubject.label

TargetSubject.label()

Returns a `LabelSubject` for the target's label value.

Method: TargetSubject.label

TargetSubject.output_group

TargetSubject.output_group(*name*)

Returns a DepsetFileSubject of the files in the named output group.

Method: TargetSubject.output_group

PARAMETERS ¶

name¶

(`str`) an output group name. If it isn't present, an error is raised.

RETURNS ¶

DepsetFileSubject of the named output group.

TargetSubject.provider

TargetSubject.provider(*provider_key*, *factory*=None)

Returns a subject for a provider in the target.

Method: TargetSubject.provider

PARAMETERS ¶

provider_key¶

The provider key to create a subject for

factory¶

(*default* `None`) optional callable. The factory function to use to create the subject for the found provider. Required if the provider key is not an inherently supported provider. It must have the following signature: `def factory(value, /, *, meta)`. Additional types of providers can be pre-registered by using the `provider_subject_factories` arg of `analysis_test`.

RETURNS ¶

A subject wrapper of the provider value.

TargetSubject.action_generating

TargetSubject.action_generating(*short_path*)

Get the single action generating the given path.

Method: TargetSubject.action_generating

NOTE: in order to use this method, the target must have the `TestingAspectInfo` provider (added by the `testing_aspect` aspect.)

PARAMETERS ¶

short_path¶

(`str`) the output's short_path to match. The value is formatted using *`format_str`*, so its template keywords can be directly passed.

RETURNS ¶

`ActionSubject` for the matching action. If no action is found, or more than one action matches, then an error is raised.

TargetSubject.action_named

TargetSubject.action_named(*mnemonic*)

Get the single action with the matching mnemonic.

Method: TargetSubject.action_named

NOTE: in order to use this method, the target must have the [`TestingAspectInfo`] provider (added by the [`testing_aspect`] aspect.)

PARAMETERS ¶

mnemonic¶

(`str`) the mnemonic to match

RETURNS ¶

`ActionSubject`. If no action matches, or more than one action matches, an error is raised.

TargetSubject.attr

TargetSubject.attr(*name*, *factory*=None)

Gets a subject-wrapped value for the named attribute.

Method: TargetSubject.attr

NOTE: in order to use this method, the target must have the `TestingAspectInfo` provider (added by the `testing_aspect` aspect.)

PARAMETERS ¶

name¶

(`str`) the attribute to get. If it's an unsupported attribute, and no explicit factory was provided, an error will be raised.

factory¶

(*default* `None`) (callable) function to create the returned subject based on the attribute value. If specified, it takes precedence over the attributes that are inherently understood. It must have the following signature: `def factory(value, *, meta)`, where `value` is the value of the attribute, and `meta` is the call chain metadata.

RETURNS ¶

A Subject-like object for the given attribute. The particular subject type returned depends on attribute and `factory` arg. If it isn't know what type of subject to use for the attribute, an error is raised.

3.7.21 Truth

Truth-style asserts for Bazel's Starlark.

These asserts follow the Truth-style way of performing assertions. This basically means the actual value is wrapped in a type-specific object that provides type-specific assertion methods. This style provides several benefits: * A fluent API that more directly expresses the assertion * More egonomic assert functions * Error messages with more informative context * Promotes code reuses at the type-level.

For more detailed documentation, see the docs on GitHub.

Basic usage

NOTE: This example assumes usage of [`rules_testing`]'s [`analysis_test`] framework, but that framework is not required.

```
def foo_test(env, target):
    subject = env.expect.that_target(target)
    subject.runfiles().contains_at_least(["foo.txt"])
    subject.executable().equals("bar.exe")

    subject = env.expect.that_action(...)
    subject.contains_at_least_args(...)
```

matching.contains

matching.contains(*contained*)

Match that `contained` is within the to-be-matched value.

This is equivalent to: `contained in to_be_matched`. See `_match_is_in` for the reversed operation.

PARAMETERS ¶

contained¶

the value that to-be-matched value must contain.

RETURNS ¶

[`Matcher`] (see `_match_custom`).

matching.custom

matching.custom(*desc*, *func*)

Wrap an arbitrary function up as a Matcher.

Method: Matcher.new

`Matcher` struct attributes:

- `desc`: (`str`) a human-friendly description
- `match`: (callable) accepts 1 positional arg (the value to match) and returns `bool` (`True` if it matched, `False` if not).

PARAMETERS ¶

desc¶

(`str`) a human-friendly string describing what is matched.

func¶

(callable) accepts 1 positional arg (the value to match) and returns `bool` (`True` if it matched, `False` if not).

RETURNS ¶

[`Matcher`] (see above).

matching.equals_wrapper

matching.equals_wrapper(*value*)

Match that a value equals `value`, but use `value` as the `desc`.

This is a helper so that simple equality comparisons can re-use predicate based APIs.

PARAMETERS ¶

value¶

object, the value that must be equal to.

RETURNS ¶

[`Matcher`] (see `_match_custom()`), whose description is `value`.

matching.file_basename_contains

matching.file_basename_contains(*substr*)

Match that a a `File.basename` string contains a substring.

PARAMETERS ¶

substr¶

(`str`) the substring to match.

RETURNS ¶

[`Matcher`] (see `_match_custom()`).

matching.file_basename_equals

matching.file_basename_equals(*value*)

Match that a `File.basename` string equals `value`.

PARAMETERS ¶

value¶

(`str`) the basename to match.

RETURNS ¶

[`Matcher`] instance

matching.file_path_matches

matching.file_path_matches(*pattern*)

Match that a `File.path` string matches a glob-style pattern.

PARAMETERS ¶

pattern¶

(`str`) the pattern to match. "*" can be used to denote "match anything".

RETURNS ¶

[`Matcher`] (see `_match_custom`).

matching.file_extension_in

matching.file_extension_in(*values*)

Match that a `File.extension` string is any of `values`.

See also: `file_path_matches` for matching extensions that have multiple parts, e.g. `*.tar.gz` or `*.so.*`.

PARAMETERS ¶

values¶

(`list` of `str`) the extensions to match.

RETURNS ¶

[`Matcher`] instance

matching.is_in

matching.is_in(*values*)

Match that the to-be-matched value is in a collection of other values.

This is equivalent to: `to_be_matched in values`. See `_match_contains` for the reversed operation.

PARAMETERS ¶

values¶

The collection that the value must be within.

RETURNS ¶

[`Matcher`] (see `_match_custom()`).

matching.never

matching.never(*desc*)

A matcher that never matches.

This is mostly useful for testing, as it allows preventing any match while providing a custom description.

PARAMETERS ¶

desc¶

(str) human-friendly string.

RETURNS ¶

[Matcher] (see _match_custom).

matching.str_endswith

matching.str_endswith(*suffix*)

Match that a string contains another string.

PARAMETERS ¶

suffix¶

(str) the suffix that must be present

RETURNS ¶

[Matcher] (see _match_custom).

matching.str_matches

matching.str_matches(*pattern*)

Match that a string matches a glob-style pattern.

PARAMETERS ¶

pattern¶

(str) the pattern to match. * can be used to denote "match anything". There is an implicit * at the start and end of the pattern.

RETURNS ¶

[Matcher] object.

matching.str_startswith

matching.str_startswith(*prefix*)

Match that a string contains another string.

PARAMETERS ¶

prefix¶

(str) the prefix that must be present

RETURNS ¶

[Matcher] (see _match_custom).

matching.is_matcher

matching.is_matcher(*obj*)

PARAMETERS ¶

obj¶

undocumented

subjects.bool

subjects.bool(*value*, *meta*)

Creates a "BoolSubject" struct.

Method: BoolSubject.new

PARAMETERS ¶

value¶

(`bool`) the value to assert against.

meta¶

(*ExpectMeta*) the metadata about the call chain.

RETURNS ¶

A *BoolSubject*.

subjects.collection

subjects.collection(*values*, *meta*, *container_name*="values", *sortable*=True, *element_plural_name*="elements")

Creates a "CollectionSubject" struct.

Method: CollectionSubject.new

Public Attributes:

- `actual`: The wrapped collection.

PARAMETERS ¶

values¶

([`collection`]) the values to assert against.

meta¶

(*ExpectMeta*) the metadata about the call chain.

container_name¶

(*default* *"values"*) (`str`) conceptual name of the container.

sortable¶

(*default* *True*) (`bool`) True if output should be sorted for display, False if not.

element_plural_name¶

(*default* *"elements"*) (`str`) the plural word for the values in the container.

RETURNS ¶
CollectionSubject.

subjects.default_info

subjects.default_info(*info*, *meta*)

Creates a `DefaultInfoSubject`

PARAMETERS ¶

info¶

([`DefaultInfo`]) the DefaultInfo object to wrap.

meta¶

(*ExpectMeta*) call chain information.

RETURNS ¶
[`DefaultInfoSubject`] object.

subjects.depset_file

subjects.depset_file(*files*, *meta*, *container_name*="depset", *element_plural_name*="files")

Creates a DepsetFileSubject asserting on `files`.

Method: DepsetFileSubject.new

PARAMETERS ¶

files¶

(`depset` of `File`) the values to assert on.

meta¶

(*ExpectMeta*) of call chain information.

container_name¶

(*default `"depset"`*) (`str`) conceptual name of the container.

element_plural_name¶

(*default `"files"`*) (`str`) the plural word for the values in the container.

RETURNS ¶
DepsetFileSubject object.

subjects.dict

subjects.dict(*actual*, *meta*, *container_name*="dict", *key_plural_name*="keys")

Creates a new `DictSubject`.

Method: DictSubject.new

PARAMETERS ¶

actual¶

(`dict`) the dict to assert against.

meta

(*ExpectMeta*) of call chain information.

container_name

(*default "dict"*) (str) conceptual name of the dict.

key_plural_name

(*default "keys"*) (str) the plural word for the keys of the dict.

RETURNS
New DictSubject struct.

subjects.file

subjects.file(*file*, *meta*)

Creates a FileSubject asserting against the given file.

Method: FileSubject.new

PARAMETERS

file

(File) the file to assert against.

meta

(*ExpectMeta*)

RETURNS
FileSubject object.

subjects.int

subjects.int(*value*, *meta*)

Create an "IntSubject" struct.

Method: IntSubject.new

PARAMETERS

value

(optional [int]) the value to perform asserts against may be None.

meta

(*ExpectMeta*) the meta data about the call chain.

RETURNS
IntSubject.

subjects.label

subjects.label(*label*, *meta*)

Creates a new `LabelSubject` for asserting `Label` objects.

Method: LabelSubject.new

PARAMETERS ¶

label¶

(`Label`) the label to check against.

meta¶

(*`ExpectMeta`*) the metadata about the call chain.

RETURNS ¶

`LabelSubject`.

subjects.runfiles

subjects.runfiles(*runfiles*, *meta*, *kind*=None)

Creates a "RunfilesSubject" struct.

Method: RunfilesSubject.new

PARAMETERS ¶

runfiles¶

([`runfiles`]) the runfiles to check against.

meta¶

(*`ExpectMeta`*) the metadata about the call chain.

kind¶

(*default `None`*) (optional `str`) what type of runfiles they are, usually "data" or "default". If not known or not applicable, use None.

RETURNS ¶

`RunfilesSubject` object.

subjects.str

subjects.str(*actual*, *meta*)

Creates a subject for asserting strings.

Method: StrSubject.new

PARAMETERS ¶

actual¶

(`str`) the string to check against.

meta¶

(*`ExpectMeta`*) of call chain information.

RETURNS ¶
StrSubject object.

subjects.struct

subjects.struct(*actual*, *meta*, *attrs*)

Creates a `StructSubject`, which is a thin wrapper around a `struct`.

PARAMETERS ¶

actual¶

(`struct`) the struct to wrap.

meta¶

(*ExpectMeta*) object of call context information.

attrs¶

(`dict` of `str` to [`callable`]) the functions to convert attributes to subjects. The keys are attribute names that must exist on `actual`. The values are functions with the signature `def factory(value, *, meta)`, where `value` is the actual attribute value of the struct, and `meta` is an *ExpectMeta* object.

RETURNS ¶
StructSubject object, which is a struct with the following shape: * `actual` attribute, the underlying struct that was wrapped. * A callable attribute for each `attrs` entry; it takes no args and returns what the corresponding factory from `attrs` returns.

subjects.target

subjects.target(*target*, *meta*)

Creates a subject for asserting Targets.

Method: TargetSubject.new

Public attributes:

- `actual`: The wrapped `Target` object.

PARAMETERS ¶

target¶

(`Target`) the target to check against.

meta¶

(*ExpectMeta*) metadata about the call chain.

RETURNS ¶
TargetSubject object

truth.expect

truth.expect(*env*)

Wrapper around `env`.

This is the entry point to the Truth-style assertions. Example usage: expect = expect(env) expect.that_action(action).contains_at_least_args(...)

The passed in `env` object allows optional attributes to be set to customize behavior. Usually this is helpful for testing. See `_fake_env()` in truth_tests.bzl for examples.

- `fail`: callable that takes a failure message. If present, it will be called instead of the regular `Expect.add_failure` logic.
- `get_provider`: callable that takes 2 positional args (target and provider) and returns the found provider or fails.
- `has_provider`: callable that takes 2 positional args (a `Target` and a [`provider`]) and returns `bool` (`True` if present, `False` otherwise) or fails.

PARAMETERS ¶

env¶

unittest env struct, or some approximation. There are several attributes that override regular behavior; see above doc.

RETURNS ¶

Expect object

3.7.22 Util

Various utilities to aid with testing.

force_exec_config

force_exec_config(*name*, *tools*=[])

Rule to force arbitrary targets to `cfg=exec` so they can be tested when used as tools.

ATTRIBUTES ¶

name¶

(required Name) A unique name for this target.

tools¶

(optional list of labels, default `[]`) A list of tools to force into the exec config

(*name*, *tools*=[])

Rule to force arbitrary targets to `cfg=exec` so they can be tested when used as tools.

ATTRIBUTES ¶

name¶

(required Name) A unique name for this target.

tools¶

(optional list of labels, default `[]`) A list of tools to force into the exec config

TestingAspectInfo

TestingAspectInfo(*attrs*, *actions*, *vars*, *bin_path*)

Details about a target-under-test useful for testing.

FIELDS ¶

attrs¶

The raw attributes of the target under test.

actions¶

The actions registered for the target under test.

vars¶

The var dict (ctx.var) for the target under text.

bin_path¶

str; the ctx.bin_dir.path value (aka execroot).

empty_file

empty_file(*name*)

Generates an empty file and returns the target name for it.

PARAMETERS ¶

name¶

str, name of the generated output file.

RETURNS ¶

str, the name of the generated output.

get_target_actions

get_target_actions(*env*)

PARAMETERS ¶

env¶

undocumented

get_target_attrs

get_target_attrs(*env*)

PARAMETERS ¶

env¶

undocumented

helper_target

helper_target(*rule*, *kwargs*)

Define a target only used as a Starlark test input.

This is useful for e.g. analysis tests, which have to setup a small graph of targets that should only be built via the test (e.g. they may require config settings the test sets). Tags are added to hide the target from `:all`, `/...`, TAP, etc.

PARAMETERS ¶

rule¶

rule-like function.

kwargs¶

Any kwargs to pass to `rule`. Additional tags will be added to hide the target.

is_file

is_file(*obj*)

Tells if an object is a File object.

PARAMETERS ¶

obj¶

undocumented

is_runfiles

is_runfiles(*obj*)

Tells if an object is a runfiles object.

PARAMETERS ¶

obj¶

undocumented

merge_kwargs

merge_kwargs(*kwargs*)

Merges multiple dicts of kwargs.

This is similar to dict.update except: * If a key's value is a list, it'll be concatenated to any existing value. * An error is raised when the same non-list key occurs more than once.

PARAMETERS ¶

kwargs¶

kwarg arg dicts to merge

RETURNS ¶

dict of the merged kwarg dics.

runfiles_map

runfiles_map(*workspace_name*, *runfiles*)

Convert runfiles to a path->file mapping.

This approximates how Bazel materializes the runfiles on the file system.

PARAMETERS ¶

workspace_name¶

str; the workspace the runfiles belong to.

runfiles¶

runfiles; the runfiles to convert to a map.

RETURNS ¶

`dict[str, optional File]` that maps the path under the runfiles root to it's backing file. The file may be None if the path came from `runfiles.empty_filenames`.

runfiles_paths

runfiles_paths(*workspace_name*, *runfiles*)

Returns the root-relative short paths for the files in runfiles.

PARAMETERS ¶

workspace_name¶

str, the workspace name (`ctx.workspace_name`).

runfiles¶

runfiles, the runfiles to convert to short paths.

RETURNS ¶

list of short paths but runfiles root-relative. e.g. 'myworkspace/foo/bar.py'.

short_paths

short_paths(*files_depset*)

Returns the `short_path` paths for a depset of files.

PARAMETERS ¶

files_depset¶

undocumented

skip_test

skip_test(*name*)

Defines a test target that is always skipped.

This is useful for tests that should be skipped if some condition, determinable during the loading phase, isn't met. The resulting target will show up as "SKIPPED" in the output.

If possible, prefer to use `target_compatible_with` to mark tests as incompatible. This avoids confusing behavior where the type of a target varies depending on loading-phase behavior.

PARAMETERS ¶

name¶

The name of the target.

util.empty_file

util.empty_file(*name*)

Generates an empty file and returns the target name for it.

PARAMETERS ¶

name¶

str, name of the generated output file.

RETURNS ¶

str, the name of the generated output.

util.helper_target

util.helper_target(*rule*, *kwargs*)

Define a target only used as a Starlark test input.

This is useful for e.g. analysis tests, which have to setup a small graph of targets that should only be built via the test (e.g. they may require config settings the test sets). Tags are added to hide the target from `:all`, `/...`, TAP, etc.

PARAMETERS ¶

rule¶

rule-like function.

kwargs¶

Any kwargs to pass to `rule`. Additional tags will be added to hide the target.

util.merge_kwargs

util.merge_kwargs(*kwargs*)

Merges multiple dicts of kwargs.

This is similar to dict.update except: * If a key's value is a list, it'll be concatenated to any existing value. * An error is raised when the same non-list key occurs more than once.

PARAMETERS ¶

kwargs¶

kwarg arg dicts to merge

RETURNS ¶

dict of the merged kwarg dics.

util.runfiles_map

util.runfiles_map(*workspace_name*, *runfiles*)

Convert runfiles to a path->file mapping.

This approximates how Bazel materializes the runfiles on the file system.

PARAMETERS ¶

workspace_name¶

str; the workspace the runfiles belong to.

runfiles¶

runfiles; the runfiles to convert to a map.

RETURNS ¶

`dict[str, optional File]` that maps the path under the runfiles root to it's backing file. The file may be None if the path came from `runfiles.empty_filenames`.

util.runfiles_paths

util.runfiles_paths(*workspace_name*, *runfiles*)

Returns the root-relative short paths for the files in runfiles.

PARAMETERS ¶

workspace_name¶

str, the workspace name (`ctx.workspace_name`).

runfiles¶

runfiles, the runfiles to convert to short paths.

RETURNS ¶

list of short paths but runfiles root-relative. e.g. 'myworkspace/foo/bar.py'.

util.short_paths

util.short_paths(*files_depset*)

Returns the `short_path` paths for a depset of files.

PARAMETERS ¶

files_depset¶

undocumented

util.skip_test

util.skip_test(*name*)

Defines a test target that is always skipped.

This is useful for tests that should be skipped if some condition, determinable during the loading phase, isn't met. The resulting target will show up as "SKIPPED" in the output.

If possible, prefer to use `target_compatible_with` to mark tests as incompatible. This avoids confusing behavior where the type of a target varies depending on loading-phase behavior.

PARAMETERS ¶

name¶

The name of the target.

recursive_testing_aspect

ASPECT ATTRIBUTES

Name	Type
*	String

ATTRIBUTES

Name	Description	Type	Mandatory	Default
name	A unique name for this target.	Name	required	

testing_aspect

ASPECT ATTRIBUTES

ATTRIBUTES

Name	Description	Type	Mandatory	Default
name	A unique name for this target.	Name	required	