Rotest Documentation

Release 7.3.0

gregoil

Jun 02, 2019

Contents

CHAPTER 1

Basic

1.1 Rotest

Watch the demo

Rotest is a resource oriented testing framework, for writing system or integration tests.

Rotest is based on Python's *unittest* module and on the Django framework. It enables defining simple abstracted components in the system, called resources. The resources may be DUT (devices under test) or they may help the test process. The tests look very much like tests written using the builtin module *unittest*.

1.1.1 Why Use Rotest?

- Allowing teams to share resources without interfering with one another.
- Easily abstracting automated components in the system.
- Lots of useful features: multiprocess, filtering tests, variety of output handlers (and the ability to create custom ones), and much more.

1.1.2 Examples

For a complete step-by-step explanation about the framework, you can read our documentation at Read The Docs. If you just want to see how it looks, read further.

For our example, let's look at an example for a `Calculator` resource:

```
import os
import rpyc
from django.db import models
from rotest.management import base_resource
from rotest.management.models import resource_data


class CalculatorData(resource_data.ResourceData):
    class Meta:
        app_label = "resources"

    ip_address = models.IPAddressField()


class Calculator(base_resource.BaseResource):
    DATA_CLASS = CalculatorData

    PORT = 1357
    EXECUTABLE_PATH = os.path.join(os.path.expanduser("~"),
                                   "calc.py")

    def connect(self):
        self._rpyc = rpyc.classic.connect(self.data.ip_address,
                                          self.PORT)

    def calculate(self, expression):
        result = self._rpyc.modules.subprocess.check_output(
            ["python", self.EXECUTABLE_PATH, expression])
        return int(result.strip())

    def finalize(self):
        if self._rpyc is not None:
            self._rpyc.close()
            self._rpyc = None
```

The `CalculatorData` class is a standard Django model that exposes IP address of the calculator machine through the data attribute. Also, we're using *rpyc* for automating the access to those machines. Except from that, it's easy to notice how the *connect* method is making the connection to the machine, and how the *finalize* method is cleaning afterwards.

Now, an example for a test:

```
from rotest import main
from rotest.core import TestCase


class SimpleCalculationTest(TestCase):
    calculator = Calculator()

    def test_simple_calculation(self):
        self.assertEqual(self.calculator.calculate("1+2"), 3)


if __name__ == "__main__":
    main()
```

The test may include the `setUp` and `tearDown` methods of *unittest* as well, and it differs only in the request for

resources.

Following, those are the options exposed when running the test:

```
$ rotest -h
Run tests in a module or directory.

Usage:
    rotest [<path>...] [options]

Options:
    -h,  --help
            Show help message and exit.
    --version
            Print version information and exit.
    -c <path>, --config <path>
            Test configuration file path.
    -s, --save-state
            Enable saving state of resources.
    -d <delta-iterations>, --delta <delta-iterations>
            Enable run of failed tests only - enter the number of times the
            failed tests should be run.
    -p <processes>, --processes <processes>
            Use multiprocess test runner - specify number of worker
            processes to be created.
    -o <outputs>, --outputs <outputs>
            Output handlers separated by comma.
    -f <query>, --filter <query>
            Run only tests that match the filter expression,
            e.g. 'Tag1* and not Tag13'.
    -n <name>, --name <name>
            Assign a name for current launch.
    -l, --list
            Print the tests hierarchy and quit.
    -F, --failfast
            Stop the run on first failure.
    -D, --debug
            Enter ipdb debug mode upon any test exception.
    -S, --skip-init
            Skip initialization and validation of resources.
    -r <query>, --resources <query>
            Specify resources to request by attributes,
            e.g. '-r res1.group=QA,res2.comment=CI'.
```

1.2 Getting Started

Using Rotest is very easy! We'll guide you with a plain and simple tutorial, or you can delve into each step separately.

1.2.1 Installation

Installing Rotest is very easy. The recommended way is using pip:

```
$ pip install rotest
```

If you prefer to get the latest features, you can install Rotest from source:

```
$ git clone https://github.com/gregoil/rotest
$ cd rotest
$ python setup.py install
```

1.2.2 Basic Usage

In this tutorial you'll learn:

- What are the building blocks of Rotest.
- How to create a Rotest project.
- How to run tests.

The Building Blocks of Rotest

Rotest is separated into several component types, each performs its specific tasks. Here is a brief explanation of the components:

- `rotest.core.TestCase`: The most basic runnable unit. Just like `unittest.TestCase`, it defines the actions and assertions that should be performed to do the test. For example:

```
from rotest.core import TestCase


class MyCase(TestCase):
    def test_something(self):
        result = some_function()
        self.assertEqual(result, some_value)
```

- `rotest.core.TestSuite`: Again, a known concept from the `unittest` module. It aggregates tests, to make a semantic separation between them. This way, you can hold a bunch of tests and run them as a set. A `rotest.core.TestSuite` can hold each of the following:
  - `rotest.core.TestCase` classes.
  - `rotest.core.TestSuite` classes.
  - The more complex concept of `rotest.core.TestFlow` classes.

```
from rotest.core import TestSuite


class MySuite(TestSuite):
    components = [TestCase1,
                  TestCase2,
                  OtherTestSuite]
```

Creating a Rotest Project

Rotest has a built in a client-server infrastructure, for a good reason. There must be someone who can distribute resources between tests, that are being run by several developers or testers. Thus, there must be a server that have a database of all the instances. Rotest uses the infrastructure of Django, to define this database, and to make use of the Django's admin frontend to enable changing it.

First, create a Django project, using:

```
$ django-admin startproject rotest_demo
$ cd rotest_demo
```

You'll end up with the following tree:

```
.
├── manage.py
└── rotest_demo
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Now add an end-point for rotest urls in the `urls.py` file:

```
from django.contrib import admin
from django.conf.urls import include, url

admin.autodiscover()
urlpatterns = [
    url(r'^rotest/api/', include("rotest.api.urls")),
    url(r'^admin/', include(admin.site.urls)),
]
```

Note: Pay attention to the base url given - *rotest/api/* is the default end-point, if it is different make sure to update it in the `rotest.yml` file.

Inside it, create a file in the root directory of the project called `rotest.yml`, that includes all configuration of Rotest:

```
rotest:
    host: localhost
    api_base_url: rotest/api/
    django_settings: rotest_demo.settings
```

Pay attention to the following:

- The *rotest* keyword defines its section as the place for Rotest's configuration.
- The *host* key is how the client should contact the server. It's an IP address, or a DNS of the server. For now, both the client and server are running on the same machine., but it doesn't have to be that way.
- The *api_base_url* key is the end-point url of the rotest urls file, it must be configured for the resource management to work - default is "rotest/api/".
- The *django_settings* key is directing to the settings of the Django app, that defines all relevant Django configuration (DB configuration, installed Django applications, and so on).

Adding Tests

Let's create a test that doesn't require any resource. Create a file named `test_math.py` with the following content:

```
from rotest import main
from rotest.core import TestCase


```

(continues on next page)

(continued from previous page)

```
class AddTest(TestCase):
    def test_add(self):
        self.assertEqual(1 + 1, 2)


if __name__ == "__main__":
    main()
```

That's a very simple test, that asserts integers addition operation in Python. To run it, just do the following:

```
$ python test_math.py
    21:46:20 : Test run has started
Tests Run Started
    21:46:20 : Test AnonymousSuite_None has started running
Test AnonymousSuite Started
    21:46:20 : Running AnonymousSuite_None test-suite
    21:46:20 : Test AddTest.test_add_None has started running
Test AddTest.test_add Started
    21:46:20 : Finished setUp - Skipping test is now available
    21:46:20 : Starting tearDown - Skipping test is unavailable
    21:46:20 : Test AddTest.test_add_None ended successfully
Success: test_add (__main__.AddTest)
    21:46:20 : Test AddTest.test_add_None has stopped running
Test AddTest.test_add Finished
    21:46:20 : Test AnonymousSuite_None has stopped running
Test AnonymousSuite Finished
    21:46:20 : Test run has finished
Tests Run Finished

Ran 1 test in 0.012s

OK
  21:46:20 : Finalizing 'AnonymousSuite' test runner
  21:46:20 : Finalizing test 'AnonymousSuite'
```

Alternatively, you can skip importing and using `rotest.main()`, and use the built-in tests discoverer:

```
$ rotest test_math.py
or
$ rotest <dir to search tests in>
```

1.2.3 Adding Resources

The true power of Rotest is in its client-server infrastructure, which enables writing resource-oriented tests, running a dedicated server to hold all resources, and enabling clients run tests.

In this tutorial, you'll learn:

- How to create a resource class.
- How to run the server, that acts as a resource manager.

Creating a Resource Class

In the root of your project, create a new Django application:

```
$ django-admin startapp resources
```

You'll see a new directory named `resources`, in the following structure:

```
.
├── manage.py
├── resources
│   ├── admin.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── rotest_demo
│   ├── __init__.py
│   ├── __init__.pyc
│   ├── settings.py
│   ├── settings.pyc
│   ├── urls.py
│   └── wsgi.py
├── rotest.yml
└── test_math.py
```

Don't forget to add the new application as well as `rotest` to the `INSTALLED_APPS` configuration in the `rotest_demo/settings.py` file:

```
...

INSTALLED_APPS = (
    'rotest.core',
    'rotest.management',
    'resources',
    'django.contrib.admin',
    'django.contrib.auth',
    ...
)
```

We're going to write a simple resource of a calculator. Edit the `resources/models.py` file to have the following content:

```
from django.db import models
from rotest.management.models.resource_data import ResourceData


class CalculatorData(ResourceData):
    class Meta:
        app_label = "resources"

    ip_address = models.IPAddressField()
```

The `CalculatorData` class is the database definition of the Calculator resource. It defines any characteristics it has, as oppose to behaviour it may have. It's also recommended adding it to the Django admin panel. Edit the content of the `resources/admin.py` file:

```
from rotest.management.admin import register_resource_to_admin

```

(continues on next page)

(continued from previous page)

```
from . import models

register_resource_to_admin(models.CalculatorData, attr_list=['ip_address'])
```

Let's continue to write the Calculator resource, which exposes a simple calculation action. Edit the file `resources/resources.py`:

```
import rpyc
from rotest.management.base_resource import BaseResource

from .models import CalculatorData


class Calculator(BaseResource):
    DATA_CLASS = CalculatorData

    PORT = 1357

    def connect(self):
        super(Calculator, self).connect()
        self._rpyc = rpyc.classic.connect(self.data.ip_address, self.PORT)

    def finalize(self):
        super(Calculator, self).finalize()
        if self._rpyc is not None:
            self._rpyc.close()
            self._rpyc = None

    def calculate(self, expression):
        return self._rpyc.eval(expression)
```

Note the following:

- *Rotest* expects a `resources.py` or `resources/__init__.py` file to be present in your resources application, in which all your *BaseResource* classes would be written or imported, much like how *Django* expects a `models.py` in for the models.
- This example uses the `RPyC` module, which can be installed using:

  ```
  $ pip install rpyc
  ```

- The `Calculator` class inherits from `rotest.management.base_resource.BaseResource`.
- The previously declared class `CalculatorData` is referenced in this class.
- Two methods are used to set up and tear down the connection to the resource: `rotest.management.base_resource.BaseResource.connect()` and `rotest.management.base_resource.BaseResource.finalize()`.

The methods of BaseResource that can be overridden:

- **connect()** - Always called at the start of the resource's setup process, override this method to start the command interface to your resource, e.g. setting up a SSH connection, creating a Selenium client, etc.
- **validate()** - Called after `connect` if the `skip_init` flag was off (which is the default). This method should return *False* if further initialization is needed to set up the resource, or *True* if it is ready to work as it is. The default `validate` method always returns *False*, prompting the resource's initialization process after `connect` (see next method).

- **initialize()** - Called after `connect` if the `skip_init` flag was off (which is the default) and `validate` returned *False* (which is also the default). Override this method to further prepare the resource for work, e.g. installing versions and files, starting up processes, etc.
- **finalize()** - Called when the resource is released, override this method to to clean temporary files, shut down processes, destroy the remote connection, etc.
- **store_state(state_dir_path)** - Called after the teardown of a test, but only if `save_state` flag was on (which is *False* by default) and the test ended in an error or a failure. The directory path which is passed to this method is a dedicated folder inside the test's working directory. Override this method to create a snapshot of the resource's state for debugging purposes, e.g. copying logs, etc.

Running the Resource Management Server

First, let's initialize the database with the following Django commands:

```
$ python manage.py makemigrations
Migrations for 'resources':
  0001_initial.py:
    - Create model CalculatorData
$ python manage.py migrate
Operations to perform:
  Apply all migrations: core, management, sessions, admin, auth, contenttypes,␣
↪resources
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying management.0001_initial... OK
  Applying management.0002_auto_20150224_1427... OK
  Applying management.0003_add_isusable_and_comment... OK
  Applying management.0004_auto_20150702_1312... OK
  Applying management.0005_auto_20150702_1403... OK
  Applying management.0006_delete_projectdata... OK
  Applying management.0007_baseresource_group... OK
  Applying management.0008_add_owner_reserved_time... OK
  Applying management.0009_initializetimeoutresource... OK
  Applying management.0010_finalizetimeoutresource... OK
  Applying management.0011_refactored_to_resourcedata... OK
  Applying management.0012_delete_previous_resources... OK
  Applying core.0001_initial... OK
  Applying core.0002_auto_20170308_1248... OK
  Applying management.0013_auto_20170308_1248... OK
  Applying resources.0001_initial... OK
  Applying sessions.0001_initial... OK
```

The first command creates a migrations file, that orders changing the database schemas or contents. The second command changes the database according to those orders. If the database does not already exist, it creates it.

Let's run the Rotest server, using the **`rotest server`** command:

```
$ rotest server

Performing system checks...

System check identified no issues (0 silenced).
May 23, 2018 - 20:05:28
```

(continues on next page)

(continued from previous page)

```
Django version 1.7.11, using settings 'rotest_demo.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
```

Adding a Resource on Django Admin Panel

To sum this up, let's add a Calculator resource. Run the *createsuperuser* command to get access to the admin panel:

```
$ python manage.py createsuperuser
Username (leave blank to use 'user'): <choose a user in here>
Email address: <choose your email address>
Password: <type in your password>
Password (again): <type password again>
Superuser created successfully.
```

Now, Just enter the Django admin panel (via http://127.0.0.1:8000/admin), access it using the above credentials, and add a resource with the name `calc` and a local IP address like `127.0.0.1`:

1.2.4 Rotest Usage

Rotest Shell

The *rotest shell* is an extension of an *IPython* environment meant to work with resources and tests.

It creates a resources client, starts a log-to-screen pipe, automatically imports resources, and provides basic functions to run tests.

Using the shell:

```
$ rotest shell
Creating client
Done! You can now lock resources and run tests, e.g.
    resource1 = ResourceClass.lock(skip_init=True, name='resource_name')
    resource2 = ResourceClass.lock(name='resource_name', config='config.json')
    shared_data['resource'] = resource1
    run_block(ResourceBlock, parameter=5)
    run_block(ResourceBlock.params(parameter=6), resource=resource2)

Python 2.7.15 (default, Jun 27 2018, 13:05:28)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: calc = Calculator.lock()
06:08:34 : Requesting resources from resource manager
06:08:34 : Locked resources [Calculator(CalculatorData('calc'))]
06:08:34 : Setting up the locked resources
06:08:34 : Resource 'shell_resource' work dir was created under '~/.rotest'
06:08:34 : Connecting resource 'calc'
06:08:34 : Initializing resource 'calc'
```

(continues on next page)

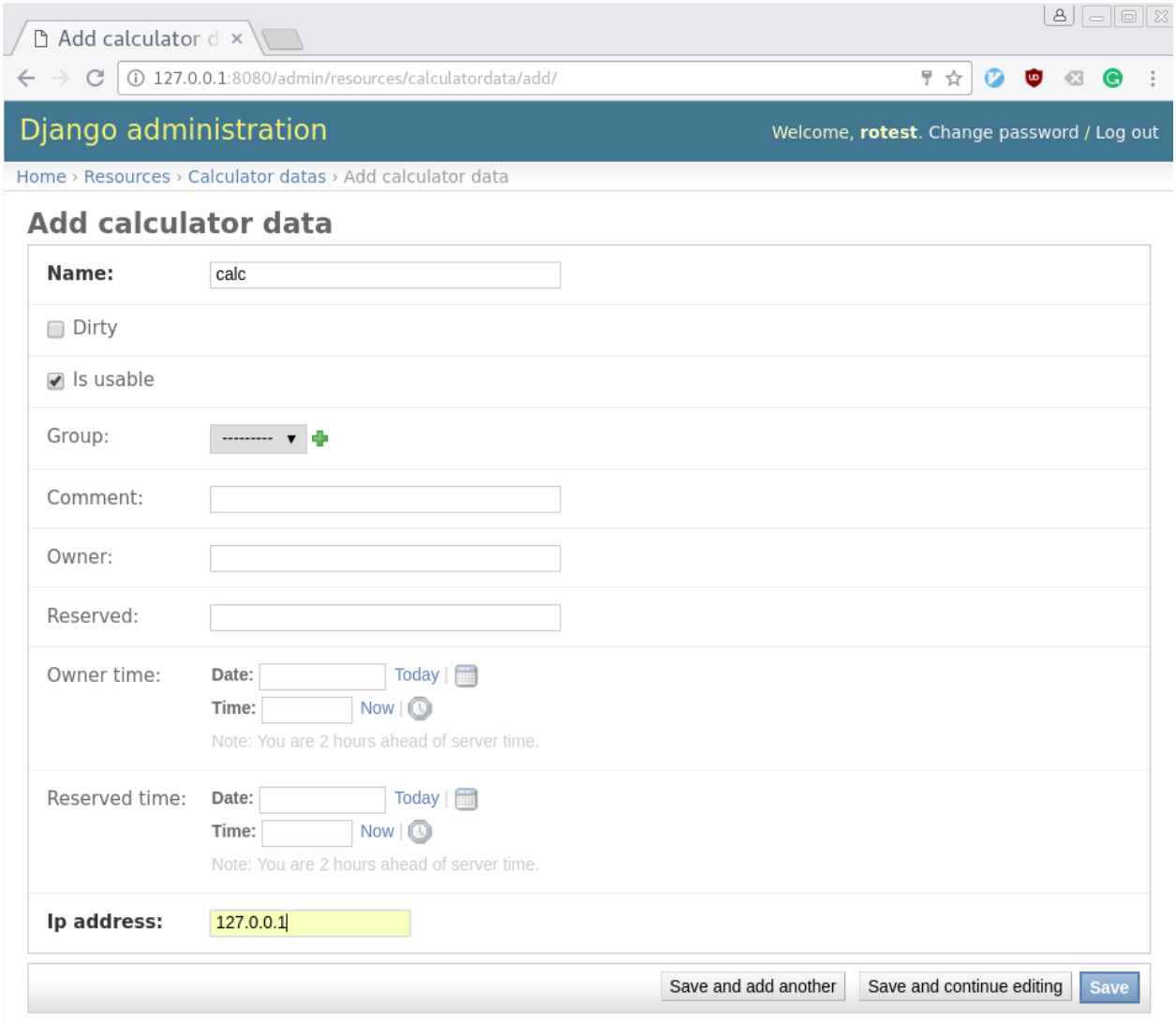

Fig. 1: Adding a resource via Django admin

(continued from previous page)

```
06:08:34 : Resource 'calc' validation failed
06:08:34 : Initializing resource 'calc'
06:08:34 : Resource 'calc' was initialized

In [2]: print calc.calculate("1 + 1")
2
```

All *BaseResources* have a *lock* method that can be used in the shell and in scripts, which requests and initializes resources, returning a resource that's ready for work.

You can add more startup commands to the rotest shell via the entry-point *shell_startup_commands*. For more information, see *Configurations*.

Writing a Resource-Based Test

In this section, we are going to add our resource to our existing test. The first thing we need to do, is setting up our resource named `calc`. We need to run the RPyC server of the calculator, using the following command:

```
$ rpyc_classic.py --port 1357
INFO:SLAVE/1357:server started on [0.0.0.0]:1357
```

This way, we have a way to communicate to our resource, which is running on our local computer (or may run on other computer, assuming you've set the corresponding IP address in the Django admin).

Now, let's change the previously written module `test_math.py` with the following content:

```
from rotest.core import TestCase

from resources.resources import Calculator


class AddTest(TestCase):
    calc = Calculator.request()

    def test_add(self):
        result = self.calc.calculate("1 + 1")
        self.assertEqual(result, 2)
```

We can request resources in the test's scope in two different ways.

- As shown in the example, write a request of the format:

  ```
  <request_name> = <resource_class>.request(<request_filters or service_parameters>)
  ```

  The optional `request filters` (in case of a resource that has data) are of the same syntax as the options passed to Django models `<Model>.objects.filter()` method, and can help you make the resource request of the test more specific, e.g.

  ```
  calc = Calculator.request(name='calc')
  ```

  If the resource doesn't point to `DATA_CLASS` (is None) then the resource is a service, and `request_filters` become initialization parameters.

- [Deprecated] Overriding the `resources` field and using `rotest.core.request` instances:

  ```
  resources = [<request1>, <request2>, ...]
  ```

where each request is of the format

request(<request_name>, <resource_class>, <request_filters or service_parameters>)

where the parameters mean the same as in the previous requesting method.

- Dynamic requests (during the test-run)

  In the test method, you can call `self.request_resources([<request1>, <request2>, ...])`

  The requests are instances of `rotest.core.request`, as in the previous method.

> **Warning:** The method for declaring test resource and sub-resources has changed since version 6.0.0.
>
> The previous method didn't use the *request* classmethod, and instead used the constructor, e.g. calc = Calculator().
>
> That form is no longer supported!

Now, let's run the test:

```
$ rotest test_math.py
AnonymousSuite
  AddTest.test_add ... OK

Ran 1 test in 0.160s

OK
```

Test event methods

Test result events you can use in Rotest:

- *self.fail(<message>)*, *self.skip(<message>)* as in `unittest`.
- All failure events using *assert<X>*, as in `unittest`.
- *expect<X>* methods (a new concept) - for cases where you want to fail the test but don't want the action to break the test flow.

  `expect` only registers the failures (if there are any) but stays in the same scope, allowing for more testing actions in the same single test. E.g.

```python
from rotest.core import TestCase

from resources.resources import Calculator


class AddTest(TestCase):
      calc = Calculator()

    def test_add(self):
        self.expectEqual(self.calc.calculate("1 + 1"), 2)
        self.expectEqual(self.calc.calculate("1 + 2"), 2)
        self.expectEqual(self.calc.calculate("1 + 3"), 2)
```

  In the above example `AddTest` will have 2 failures to the same run (3!=2 and 4!=2).

  It is recommended to use `expect` to test different side-effects of the same scenario, like different side effects of the same action, but you can use it any way you please.

There is an `expect` method equivalent for every `assert` method, e.g. `expectEqual` and `expectIsNone`.

- Success events (a new concept) - When you want to register information about the test, like numeric results of actions or time measurement of actions.

  The success information will be registered into the test's metadata, like any other failure, error, or skip message, and will be visible in the DB, excel, etc.

```
from rotest.core import TestCase

from resources.resources import Calculator


class AddTest(TestCase):
      calc = Calculator()

    def test_add(self):

        self.success("One way to register success")
        # Or
        self.addSuccess("Another way to register success")

        value = self.calc.calculate("1 + 1")
        self.expectEqual(value, 3,
                         msg="Expected value 3, got %r" % value,
                         success_msg="Value is %r, as expected" % value)
        # Or
        self.assertEqual(value, 3,
                         msg="Expected value 3, got %r" % value,
                         success_msg="Value is %r, as expected" % value)
```

1.3 Command Line Options

Let's go over the some of Rotest features, by examining the command line options.

1.3.1 Server Options

You can run the server using command **`rotest server`**. The command by default runs Django's server with the port supplied in the rotest.yml file, defaults to 8000.

1.3.2 Client Options

Running tests

Running tests can be done in the following ways:

- Using the *rotest* command:

```
$ rotest [PATHS]... [OPTIONS]
```

  The command can get every path - either files or directories. Every directory will be recursively visited for finding more files. If no path was given, the current working directory will be selected by default.

- Calling the `rotest.main()` function:

```
from rotest import main
from rotest.core import TestCase


class Case(TestCase):
    def test(self):
        pass


if __name__ == "__main__":
    main()
```

Then, this same file can be ran:

```
$ python test_file.py [OPTIONS]
```

Getting Help

-h, --help
Show a help message and exit.

If you're not sure what you can do, the help options *-h* and *--help* are here to help:

```
$ rotest -h
Run tests in a module or directory.

Usage:
    rotest [<path>...] [options]

Options:
    -h,  --help
            Show help message and exit.
    --version
            Print version information and exit.
    -c <path>, --config <path>
            Test configuration file path.
    -s, --save-state
            Enable saving state of resources.
    -d <delta-iterations>, --delta <delta-iterations>
            Enable run of failed tests only - enter the number of times the
            failed tests should be run.
    -p <processes>, --processes <processes>
            Use multiprocess test runner - specify number of worker
            processes to be created.
    -o <outputs>, --outputs <outputs>
            Output handlers separated by comma.
    -f <query>, --filter <query>
            Run only tests that match the filter expression,
            e.g. 'Tag1* and not Tag13'.
    -n <name>, --name <name>
            Assign a name for current launch.
    -l, --list
            Print the tests hierarchy and quit.
    -F, --failfast
            Stop the run on first failure.
    -D, --debug
```

(continues on next page)

(continued from previous page)

```
            Enter ipdb debug mode upon any test exception, and enable
            entering debug mode on Ctrl-Pause (Windows) or Ctrl-Quit (Linux).
    -S, --skip-init
            Skip initialization and validation of resources.
    -r <query>, --resources <query>
            Specify resources to request by attributes,
            e.g. '-r res1.group=QA,res2.comment=CI'.
```

Listing, Filtering and Ordering

-l, --list
Print the tests hierarchy and quit.

-f <query>**, --filter** <query>
Run only tests that match the filter expression, e.g. "Tag1* and not Tag13".

Next, you can print a list of all the tests that will be run, using *-l* or *--list* options:

```
$ rotest some_test_file.py -l
CalculatorSuite []
|   CasesSuite []
|   |   PassingCase.test_passing ['BASIC']
|   |   FailingCase.test_failing ['BASIC']
|   |   ErrorCase.test_error ['BASIC']
|   |   SkippedCase.test_skip ['BASIC']
|   |   SkippedByFilterCase.test_skipped_by_filter ['BASIC']
|   |   ExpectedFailureCase.test_expected_failure ['BASIC']
|   |   UnexpectedSuccessCase.test_unexpected_success ['BASIC']
|   PassingSuite []
|   |   PassingCase.test_passing ['BASIC']
|   |   SuccessFlow ['FLOW']
|   |   |   PassingBlock.test_method
|   |   |   PassingBlock.test_method
|   FlowsSuite []
|   |   FailsAtSetupFlow ['FLOW']
|   |   |   PassingBlock.test_method
|   |   |   FailingBlock.test_method
|   |   |   ErrorBlock.test_method
|   |   FailsAtTearDownFlow ['FLOW']
|   |   |   PassingBlock.test_method
|   |   |   TooManyLogLinesBlock.test_method
|   |   |   FailingBlock.test_method
|   |   |   ErrorBlock.test_method
|   |   SuccessFlow ['FLOW']
|   |   |   PassingBlock.test_method
|   |   |   PassingBlock.test_method
```

You can see the tests hierarchy, as well as the tags each test has. Speaking about tags, you can apply filters on the tests to be run, or on the shown list of tests using the *-f* or *--filter* options:

```
    $ rotest some_test_file.py -f FLOW -l
    CalculatorSuite []
    |   CasesSuite []
    |   |   PassingCase.test_passing ['BASIC']
    |   |   FailingCase.test_failing ['BASIC']
```

(continues on next page)

(continued from previous page)

```
    |   |   ErrorCase.test_error ['BASIC']
    |   |   SkippedCase.test_skip ['BASIC']
    |   |   SkippedByFilterCase.test_skipped_by_filter ['BASIC']
    |   |   ExpectedFailureCase.test_expected_failure ['BASIC']
    |   |   UnexpectedSuccessCase.test_unexpected_success ['BASIC']
    |   PassingSuite []
    |   |   PassingCase.test_passing ['BASIC']
    |   |   SuccessFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   PassingBlock.test_method
    |   FlowsSuite []
    |   |   FailsAtSetupFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   FailingBlock.test_method
    |   |   |   ErrorBlock.test_method
    |   |   FailsAtTearDownFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   TooManyLogLinesBlock.test_method
    |   |   |   FailingBlock.test_method
    |   |   |   ErrorBlock.test_method
    |   |   SuccessFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   PassingBlock.test_method

The output will be colored in a similar way as above.
```

You can include boolean literals like `not`, `or` and `and` in your filter, as well as using test names and wildcards (all non-literals are case insensitive):

```
    $ rotest some_test_file.py -f "basic and not skipped*" -l
    CalculatorSuite []
    |   CasesSuite []
    |   |   PassingCase.test_passing ['BASIC']
    |   |   FailingCase.test_failing ['BASIC']
    |   |   ErrorCase.test_error ['BASIC']
    |   |   SkippedCase.test_skip ['BASIC']
    |   |   SkippedByFilterCase.test_skipped_by_filter ['BASIC']
    |   |   ExpectedFailureCase.test_expected_failure ['BASIC']
    |   |   UnexpectedSuccessCase.test_unexpected_success ['BASIC']
    |   PassingSuite []
    |   |   PassingCase.test_passing ['BASIC']
    |   |   SuccessFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   PassingBlock.test_method
    |   FlowsSuite []
    |   |   FailsAtSetupFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   FailingBlock.test_method
    |   |   |   ErrorBlock.test_method
    |   |   FailsAtTearDownFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   TooManyLogLinesBlock.test_method
    |   |   |   FailingBlock.test_method
    |   |   |   ErrorBlock.test_method
    |   |   SuccessFlow ['FLOW']
    |   |   |   PassingBlock.test_method
    |   |   |   PassingBlock.test_method
```

-O <tags>, **--order** <tags>
Order discovered tests according to this list of tags, where tests answering the first tag (which syntax is similar to a filter expression) will get higher priority, tests answering the second tag will have a secondry priority, etc.

Stopping at first failure

-F, --failfast
Stop the run on first failure.

The *-F* or *--failfast* options can stop execution after first failure:

```
$ rotest some_test_file.py --failfast
CalculatorSuite
CasesSuite
  PassingCase.test_passing ... OK
  FailingCase.test_failing ... FAIL
  Traceback (most recent call last):
    File "/home/odp/code/rotest/src/rotest/core/case.py", line 310, in test_method_
↪wrapper
      test_method(*args, **kwargs)
    File "tests/calculator_tests.py", line 34, in test_failing
      self.assertEqual(1, 2)
  AssertionError: 1 != 2


======================================================================
FAIL: FailingCase.test_failing
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/odp/code/rotest/src/rotest/core/case.py", line 310, in test_method_
↪wrapper
    test_method(*args, **kwargs)
  File "tests/calculator_tests.py", line 34, in test_failing
    self.assertEqual(1, 2)
AssertionError: 1 != 2

Ran 2 tests in 0.205s

FAILED (failures=1)
```

Debug Mode

-D, --debug
Enter ipdb debug mode upon any test exception, and enable entering debug mode on Ctrl-Pause (Windows) or Ctrl-Quit (Linux).

The *-D* or *--debug* options can enter debug mode when exceptions are raised at the top level of the test code:

```
$ rotest some_test_file.py --debug
AnonymousSuite
  FailingCase.test ...
Traceback (most recent call last):
   File "tests/some_test_file.py", line 11, in test
    self.assertEqual(self.calculator.calculate("1+1"), 3)
   File "/usr/lib64/python2.7/unittest/case.py", line 513, in assertEqual
    assertion_func(first, second, msg=msg)
```

(continues on next page)

(continued from previous page)

```
   File "/usr/lib64/python2.7/unittest/case.py", line 506, in _baseAssertEqual
    raise self.failureException(msg)
 AssertionError: 2.0 != 3
> tests/some_test_file.py(12)test()
     10     def test(self):
     11         self.assertEqual(self.calculator.calculate("1+1"), 3)
---> 12
     13
     14 if __name__ == "__main__":

ipdb> help

Documented commands (type help <topic>):
========================================
EOF     c          d        help    longlist  pinfo    raise    tbreak   whatis
a       cl         debug    ignore  n         pinfo2   restart  u        where
alias   clear      disable  j       next      pp       retry    unalias
args    commands   down     jump    p         psource  return   unt
b       condition  enable   l       pdef      q        run      until
break   cont       exit     list    pdoc      quit     s        up
bt      continue   h        ll      pfile     r        step     w
```

Once in the debugging session, you can do any of the following:

- Inspect the situation, by evaluating expressions or using commands that are supported by `ipdb`. For example: continuing the flow, jumping into a specific line, etc.
- `retry` the action, if it's a known flaky action and someone's going to take care of it soon.
- `raise` the exception, and failing the test.

Furthermore, running tests with `--debug` also overrides the break\quit signals to enable you enter debug mode whenever you like. Just press *Ctrl-* on Linux machines or *Ctrl-Pause* on Windows (f*** you, Mac users) during a test to emulate an exception.

Retrying Tests

-d <delta-iterations>, **--delta** <delta-iterations>
: Rerun test a specified amount of times until it passes.

In case you have flaky tests, you can automatically rerun a test until getting a success result. Use options *--delta* or *-d*:

```
$ rotest some_test_file.py --delta 2
AnonymousSuite
  FailingCase.test ... FAIL
  Traceback (most recent call last):
    File "rotest/src/rotest/core/case.py", line 310, in test_method_wrapper
      test_method(*args, **kwargs)
    File "some_test_file.py", line 11, in test
      self.assertEqual(self.calculator.calculate("1+1"), 3)
  AssertionError: 2.0 != 3


======================================================================
FAIL: FailingCase.test
----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "rotest/src/rotest/core/case.py", line 310, in test_method_wrapper
    test_method(*args, **kwargs)
  File "some_test_file.py", line 11, in test
    self.assertEqual(self.calculator.calculate("1+1"), 3)
AssertionError: 2.0 != 3

Ran 1 test in 0.122s

FAILED (failures=1)
AnonymousSuite
  FailingCase.test ... OK

Ran 1 test in 0.082s

OK
```

Running Tests in Parallel

-p <processes>, **--processes** <processes>
Spawn specified amount of processes to execute tests.

To optimize the running time of tests, you can use options *-p* or *--processes* to run several work processes that can run tests separately.

Any test have a `TIMEOUT` attribute (defaults to 30 minutes), and it will be enforced only when spawning at least one worker process:

```
class SomeTest(TestCase):
    # Test will stop if it exceeds execution time of an hour,
    # only when the number of processes spawned is greater or equal to 1
    TIMEOUT = 60 * 60

    def test(self):
        pass
```

Specifying Resources to Use

-r <query>, **--resources** <query>
Choose resources based on the given query.

You can run tests with specific resources, using options *--resources* or *-r*.

The request is of the form:

```
$ rotest some_test_file.py --resources <query-for-resource-1>,<query-for-resource-2>,.
↪..
```

As an example, let's suppose we have the following test:

```
class SomeTest(TestCase):
    res1 = Resource1()
    res2 = Resource2()
```

(continues on next page)

(continued from previous page)

```
    def test(self):
        ...
```

You can request resources by their names:

```
$ rotest some_test_file.py --resources res1=name1,res2=name2
```

Alternatively, you can make more complex queries:

```
$ rotest some_test_file.py --resources res1.group.name=QA,res2.comment=nightly
```

Activating Output Handlers

-o <outputs>, **--outputs** <outputs>

To activate an output handler, use options *-o* or *--outputs*, with the output handlers separated using commas:

```
$ rotest some_test_file.py --outputs excel,logdebug
```

For more about output handlers, read on *Output Handlers*.

Adding New Options

You can create new CLI options and behavior using the two entrypoints: `cli_client_parsers` and `cli_client_actions`.

For example:

```
# utils/baz.py

def add_baz_option(parser):
    """Add the 'baz' flag to the CLI options."""
    parser.add_argument("--baz", "-B", action="store_true",
                        help="The amazing Baz flag")


def use_baz_option(tests, config):
    """Print the list of tests if 'baz' is on."""
    if config.baz is True:
        print tests
```

And in your `setup.py` file inside `setup()`:

```
entry_points={
    "rotest.cli_client_parsers":
        ["baz_parser = utils.baz:add_baz_option"],
    "rotest.cli_client_actions":
        ["baz_func = utils.baz:use_baz_option"]
},
```

- Make sure it's being installed in the environment by calling

```
python setup.py develop
```

1.4 Output Handlers

Output Handlers are a great concept in Rotest. They let you take actions when certain events occurs, as a logic separated from the test's logic.

Rotest has several builtin output handlers, as well as enable making custom output handlers.

1.4.1 Dots

The most compact way to display results - using one character per test:

```
$ python some_test_file.py -o dots
.FEssxu.....FsF..FEE...

...
```

Based on the following legend:

.	Success
F	Failure
E	Error
s	Skip
x	Expected Failure
u	Unexpected Success

1.4.2 Full

If you want to just be aware of every event, use the `full` output handler:

```
$ python some_test_file.py -o full
Tests Run Started
Test CalculatorSuite Started
Test CasesSuite Started
Test PassingCase.test_passing Started
Success: test_passing (__main__.PassingCase)
Test PassingCase.test_passing Finished
Test FailingCase.test_failing Started
Failure: test_failing (__main__.FailingCase)
Traceback (most recent call last):
  File "rotest/src/rotest/core/case.py", line 310, in test_method_wrapper
    test_method(*args, **kwargs)
  File "tests/calculator_tests.py", line 34, in test_failing
    self.assertEqual(1, 2)
AssertionError: 1 != 2

Test FailingCase.test_failing Finished
Test ErrorCase.test_error Started
Error: test_error (__main__.ErrorCase)
Traceback (most recent call last):
  File "rotest/src/rotest/core/case.py", line 310, in test_method_wrapper
    test_method(*args, **kwargs)
  File "tests/calculator_tests.py", line 44, in test_error
    1 / 0
```

(continues on next page)

(continued from previous page)

```
ZeroDivisionError: integer division or modulo by zero

...
```

1.4.3 Tree

For a tree view, use:

```
$ python some_test_file.py -o tree
CalculatorSuite
  CasesSuite
    PassingCase.test_passing ... OK
    FailingCase.test_failing ... FAIL
    Traceback (most recent call last):
      File "/home/odp/code/rotest/src/rotest/core/case.py", line 310, in test_method_
↪wrapper
        test_method(*args, **kwargs)
      File "tests/calculator_tests.py", line 34, in test_failing
        self.assertEqual(1, 2)
    AssertionError: 1 != 2

    ErrorCase.test_error ... ERROR
    Traceback (most recent call last):
      File "/home/odp/code/rotest/src/rotest/core/case.py", line 310, in test_method_
↪wrapper
        test_method(*args, **kwargs)
      File "tests/calculator_tests.py", line 44, in test_error
        1 / 0
    ZeroDivisionError: integer division or modulo by zero

...
```

1.4.4 Logs

To see the logs while running the tests, use `logdebug` or `loginfo`. Additionally, you can use `pretty` for an easier to read logging system. As expected, `logdebug` will print every log record with level which is higher or equal to `DEBUG` (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`), whereas `loginfo` will print every log record with level which is higher or equal to `INFO` (`INFO`, `WARNING`, `ERROR`, `CRITICAL`).

1.4.5 Excel

Sometimes, you want to have a better visualization of the results. Rotest can output the results into a human-readable `results.xls` file, which can be sent via email for instance. The relevant option is `-o excel`.

This artifact is saved in the working directory of Rotest. For more about this location, see *Configurations*.

1.4.6 Remote

When adding `remote` to the list of output handlers, all test events and results are saved in the remote (server's) database, which enables keeping tests run history. Furthermore, tests skip delta filtering (`--delta` run option) queries the remote database to see which tests already passed.

1.4.7 DB

The `db` handler behaves the same as `remote` handler, only uses a local DB (which should be defined in your project's `settings.py` file)

1.4.8 Artifact

This handler saves the working directory of the tests into a ZIP file, which might be useful for keeping important runs' logs and other files for future debugging or evaluation.

Those artifacts are saved in the artifacts directory of Rotest. It is recommended to make this folder a shared folder between all your users. For more about this location, see *Configurations*.

1.4.9 Signature

This handler saves in the remote DB patterns for errors and failures it encounters. You can also link the signatures to issues in your bug tracking system, e.g. JIRA. In the next encounters the handler will issue a warning with the supplied link via the log. The relevant option is `-o signature`.

To see the patterns, change them, and add links - go to the admin page of the server under core/signatures.

1.5 Configurations

Rotest behaviour can be configured in the following ways:

- A configuration file called `rotest.yml` in YAML format.
- Environment variables.
- Command line arguments.

Each way has its own advantages, and should be used in different occasions: configuration file fits where some configuration should be used by any user of the code, environment variables should be specific per user or maybe more session-based, and command line arguments are relevant for a specific run.

Note: In general:

- Command line arguments take precedence over environment variables.
- Environment variables take precedence over the configuration file.
- Some configuration attributes have default values, in case there's no answer.

1.5.1 General

To use a configuration file, put any of the following path names in the project's root directory: `rotest.yml`, `rotest.yaml`, `.rotest.yml`, `.rotest.yaml`.

The configuration file is of the form:

```
rotest:
    attribute1: value1
    attribute2: value2
```

You can configure environment variables this way in Linux / Mac / any Unix machine:

```
$ export ENVIRONMENT_VARIABLE=value
```

and this way in Windows:

```
$ set ENVIRONMENT_VARIABLE=value
$ setx ENVIRONMENT_VARIABLE=value  # Set it permanently (reopen the shell)
```

1.5.2 Working Directory

ROTEST_WORK_DIR
Working directory to save artifacts to.

Rotest uses the computer's storage in order to save several artifacts. You can use the following methods:

- Define *ROTEST_WORK_DIR* to point to the path.
- Define `workdir` in the configuration file:

```
rotest:
    workdir: /home/user/workdir
```

- Use the default, which is `~/.rotest` or `%HOME%\.rotest` in Windows.

1.5.3 Host

ROTEST_HOST
DNS or IP address to the Rotest's server.

Rotest is built on a client-server architecture. To define the relevant server that the client should contact with, use the following methods:

- Define *ROTEST_HOST* to point to the server DNS or IP address.
- Define `host` in the configuration file:

```
rotest:
    host: rotestserver
```

- Use the default, which is `localhost`.

1.5.4 Port

ROTEST_SERVER_PORT
Port for the Django server, to be used for communication with clients.

To define the relevant server's port that will be opened, and the port clients will communicate with, use the following methods:

- Define *ROTEST_SERVER_PORT* with the desired port.
- Define `port` in the configuration file:

```
rotest:
    port: 8585
```

- Use the default, which is `8000`.

1.5.5 Resource Request Timeout

ROTEST_RESOURCE_REQUEST_TIMEOUT
Amount of time to wait before deciding that no resource is available.

Rotest's server distributes resources to multiple clients. Sometimes, a client cannot get some of the resources at the moment, so the server returns an answer that there's no resource available. This amount of time is configurable via the following methods:

- Define *ROTEST_RESOURCE_REQUEST_TIMEOUT* with the number of seconds to wait before giving up on waiting for resources.
- Define `resource_request_timeout` in the configuration file:

```
rotest:
    resource_request_timeout: 60
```

- Use the default, which is `0` (not waiting at all).

1.5.6 Django Settings Module

DJANGO_SETTINGS_MODULE
Django configuration path, in a module syntax.

Rotest is a Django library, and as such needs its configuration module, in order to write and read data about the resources from the database. Define it in the following ways:

- Define *DJANGO_SETTINGS_MODULE*.
- Define `django_settings` in the configuration file:

```
rotest:
    django_settings: package1.package2.settings
```

- There is no default value.

1.5.7 Artifacts Directory

ARTIFACTS_DIR
Rotest artifact directory.

Rotest enables saving ZIP files containing the tests and resources data, using an output handler named `artifact` (see *Output Handlers*). Define it in the following ways:

- Define *ARTIFACTS_DIR*.
- Define `artifact_dir` in the configuration file:

```
rotest:
    artifacts_dir: ~/rotest_artifacts
```

- Use the default, which is `~/.rotest/artifacts`.

1.5.8 Shell Startup Commands

`rotest shell` enables defining startup commands, to save the user the need to write them every time. The commands must be simple one-liners. Define it in the following ways:

- Define `shell_startup_commands` in the configuration file:

```
rotest:
    shell_startup_commands: ["from tests.blocks import *"]
```

- Use the default, which is `[]`.

1.5.9 Discoverer Blacklist

Rotest enables loading resources from an app, a thing that happens automatically when running `rotest shell`, but some files can / must be skipped when searching for the resources.

Also, tests discovering (via `rotest <some_dir>`) takes this blacklist into consideration.

Define it in the following ways:

- Define `discoverer_blacklist` in the configuration file:

```
rotest:
    discoverer_blacklist: ["*/scripts/*", "*static.py"]
```

- Use the default, which is `[".tox", ".git", ".idea", "setup.py"]`.

CHAPTER 2

Advanced

2.1 Complex Resources

Sometimes we want our resources to contain sub-resources or sub-services (the difference is that sub-resources have *ResourceData* models in the DB and services does not). This can easily be achieved with *Rotest*.

2.1.1 Creating sub-resource model

In case we want to create a sub-resource, to *Calcuator* for example, we first need to point to it in the *CalculatorData* model.

(Skip this part if you want a sub-service, i.e. you don't need to hold data on the sub-resource in the server's DB, like when all its data is derived from the containing resource's data)

```
from django.db import models
from rotest.management.models.resource_data import ResourceData


class SubCalculatorData(ResourceData):
    class Meta:
        app_label = "resources"

    process_id = models.IntegerField()


class CalculatorData(ResourceData):
    class Meta:
        app_label = "resources"

    ip_address = models.IPAddressField()
    sub_process = models.ForeignKey(SubCalculatorData)
```

In this example we created the *ResourceData* model for the sub-resource (like we'd do to any new resource), and pointed to it in the original *CalculatorData* model, declaring we intend to use a sub-resource here.

Don't forget to add a reference to the model and the new field in `admin.py`:

```
from rotest.management.admin import register_resource_to_admin

from . import models

register_resource_to_admin(models.SubCalculatorData, attr_list=['process_id'])
register_resource_to_admin(models.CalculatorData, attr_list=['ip_address'],
                                                  link_list=['sub_process'])
```

Note that we used the *link_list* to point to the sub-resource and not *attr-list*, since its a model and not a regular field.

Don't forget to run `makemigrations` and `migrate` again after changing the models!

2.1.2 Declaring sub-resources

Let's continue to modify the Calculator resource, where we want to add sub-resources.

For now, let's assume we already wrote the sub-resource under `resources/sub_process.py`.

Now, edit the file `resources/resources.py`:

```
import rpyc
from rotest.management.base_resource import BaseResource

from .models import CalculatorData
from .sub_process import SubProcess


class Calculator(BaseResource):
    DATA_CLASS = CalculatorData

    PORT = 1357

    sub_process = SubProcess.request(data=CalculatorData.sub_process)

    def connect(self):
        super(Calculator, self).connect()
        self._rpyc = rpyc.classic.connect(self.data.ip_address, self.PORT)

    def finalize(self):
        super(Calculator, self).finalize()
        if self._rpyc is not None:
            self._rpyc.close()
            self._rpyc = None

    def calculate(self, expression):
        return self._rpyc.eval(expression)

    def get_sub_process_id(self, expression):
        return self.sub_process.data.process_id
```

Note the following:

- Declaring the sub-resource:

  ```
  sub_process = SubProcess.request(data=CalculatorData.sub_process)
  ```

  The syntax is the same as requesting resources for a test.

We assigned the *SubCalculatorData* model instance (pointed from the containing resource's *CalculatorData*) as the `data` for out sub-resource.

Alternatively, in case *SubProcess* was a service and not a full-fledged resource, we could have passed parameters to it in a similar way:

```
sub_process = SubProcess.request(ip_address=CalculatorData.ip_address,
                                 process_id=5)
```

- The usage of the sub-resource

```
def get_sub_process_id(self, expression):
    return self.sub_process.process_id
```

Once the sub-resource or service is declared, it can be accessed from any of the containing resource's methods, using the assigned name (in this case, the declaration line name it *sub_process*).

Lastly, let's show the sub-resource under `resources/sub_process.py`:

```
from rotest.management.base_resource import BaseResource

from .models import SubCalculatorData


class SubProcess(BaseResource):
    DATA_CLASS = SubCalculatorData

    def container_calculate(self, expression):
        return self.parent.calculate(expression)

    def get_ip_address(self):
        return self.parent.data.ip_address
```

Note that we have access to the containing resource via *parent*.

This also applies when we write sub-services, which can use the parent's methods, data, and even fields (e.g. *self.parent._rpyc*).

When writing sub-resources and services, remember two things:

- Always call *super* when overriding BaseResource's methods (connect, initialize, validate, finalize, store_state), since the basic method propagate the call to sub-resources.
- It is ok to use *self.parent* and *self.<sub-resource-name>* , but mind the context. E.g. *self.parent._rpyc* in the above example is accessible from the sub-resource, but only after the `connect()` method (since firstly the sub-resource connects, and only afterwards the containing resource connects). The same applies for the other basic methods (first the sub-resources initialize, then the containing).

2.1.3 Parallel initialization

Usually, the initialization process of resources takes a long time. In order to speed things up, each resource has a `PARALLEL_INITIALIZATION` flag.

This flag defaults to *False*, but when it is set to *True* each sub-resource would be initialized in its own thread, before joining back to the containing resource for the parent custom initialization code.

To activate it, simply write in the class scope of your complex resource:

```
class Calculator(BaseResource):
    DATA_CLASS = CalculatorData

    PARALLEL_INITIALIZATION = True

    sub_resource1 = SubResource.request()
    sub_resource2 = SubResource.request()
    sub_resource3 = SubResource.request()
```

Or you can point it to a variable which you can set/unset using an entry point (see *Adding New Options* to learn how to add CLI entry points).

2.2 Blocks code architecture

2.2.1 Background

The blocks design paradigm was created to avoid code duplication and enable composing tests faster.

`TestBlock` is a building block for tests, commonly responsible for a single action or a small set of actions. It inherits from `unittest`'s `TestCase`, enabling it test-like behavior (`self.skipTest`, `self.assertEqual`, `self.fail`, etc.), and the Rotest infrastructure expands its behavior to also be function-like (to have "inputs" and "outputs").

`TestFlow` is a test composed of `TestBlock` instances (or other sub-test flows), passing them their 'inputs' and putting them together, enabling them to share data between each other. A `TestFlow` can lock resources much like Rotest's TestCase, which it passes to all the blocks under it.

The flow's final result depends on the result of the blocks under it by the following order:

- If some block had an error, the flow ends with an error.
- If some block had a failure, the flow ends with a failure.
- Otherwise, the flow succeeds.

See also `mode` in the `TestBlock`'s "Features" segment below for more information about the run mechanism of a `TestFlow`.

2.2.2 Features

TestFlow

1. `blocks`: static list or tuple of the blocks' classes of the flow. You can parametrize blocks in this section, in order to pass data to them (see *Sharing data* section or explanation in the TestBlock features section).
2. Rotest's `TestCase` features: run delta, filter by tags, running in multiprocess, TIMEOUT, etc. are available also for `TestFlow` class.

TestBlock

1. **`inputs`: define class fields and assign them to instances of BlockInput** to ask for values for the block (values are passed via `common`, `parametrize`, previous blocks passing them as `outputs`, or as requested resources of the block or its containers). You can define a default value to BlockInput to assign if non is supplied (making it an optional input). For example, defining in the block's scope

```
from rotest.core import TestBlock, BlockInput
class DemoBlock(TestBlock):
    field_name = BlockInput()
    other_field = BlockInput(default=1)
...
```

will validate that the block instance will have a value for 'field_name' before running the parent flow (and unless another value is supplied, set for the block's instance: self.other_field=1).

2. **`outputs`: define class fields and assign them to instances of BlockOutput** to share values from the instance (self) to the parent and siblings. the block automatically shares the declared outputs after teardown. For example, defining in the block's scope

```
from rotest.core import TestBlock, BlockOutput
class DemoBlock(TestBlock):
    field_name = BlockOutput()
    other_field = BlockOutput()
...
```

means declaring that the block would calculate a values for self.field_name and self.other_field and share them (which happens automatically after its teardown), so that components following the block can use those fields. Declaring inputs and outputs of blocks is not mandatory, but it's a good way to make sure that the blocks "click" together properly, and no block will be missing fields at runtime.

Common features (for both flows and blocks)

1. `resources`: you can specify resources for the test flow or block, just like in Rotest's TestCase class. The resources of a flow will automatically propagate to the components under it.
2. `common`: used to set values to blocks or sub-flows, see example in the *Sharing data* section.
3. `parametrize` (also `params`): used to pass values to blocks or sub-flows, see example in the *Sharing data* section. Note that calling `parametrize()` or `params()` doesn't actually instantiate the component, but just create a copy of the class and sends the parameters to its common (overriding previous values).
4. `mode`: this field can be defined statically in the component's class or passed to the instance using 'parametrize' (parametrized fields override class fields of blocks, since they are injected into the instance). Blocks and sub-flows can run in one of the following modes (which are defined in `rotest.core.flow_component`)
    1. `MODE_CRITICAL`: upon failure or error, end the flow's run, skipping the following components (except those with mode `MODE_FINALLY`). Use this mode for blocks or sub-flows that do actions that are mandatory for the continuation of the test.
    2. `MODE_OPTIONAL`: upon error only, end the flow's run, skipping the following components (except those with mode `MODE_FINALLY`). Use this mode for block or sub-flows that are not critical for the continuation of the test (since a failure in them doesn't stop the flow).
    3. `MODE_FINALLY`: components with this mode aren't skipped even if the flow has already failed and stopped. Upon failure or error, end the flow's run, skipping the following components (except those with mode `MODE_FINALLY`). Use this mode for example in blocks or sub-flows that do cleanup actions (which we should always attempt), much like things you would normally put in 'tearDown' of tests.
5. `request_resources`: blocks and flows can dynamically request resources, calling `request_resources(requests)` method (see Rotest tutorial and documentation for more information).

   Since those are dynamic requests, don't forget to release those resources when they are not needed by calling

```
release_resources(
    <dict of the dynamically locked resources, name: instance>)
```

Resources can be locked locally and globally in regarding to the containing flow, i.e. by locking the resources using the parent's method:

```
self.parent.request_resources(requests)
```

The parent flow and all the sibling components would also have them.

Sharing data

Sharing data between blocks (getting inputs and passing outputs) is crucial to writing simple, manageable, and independent blocks. Passing data to blocks (for them to use as 'inputs' parameters for the block's run, much like arguments for a function) can be done in one of the following methods:

- Locking resources - the resources the flow locks are injected into its components' instances (note that blocks can also lock resources, but they don't propagate them up or down). E.g. if a flow locks a resource with name 'res1', then all its components would have the field 'res1' which points to the locked resource.
- Declaring outputs - see TestBlock's `outputs` above.
- Setting initial data to the test - you can set initial data to the component and its sub-components by writing:

```
class DemoFlow(TestFlow):
    common = {'field_name': 5,
              'other_field': 'abc'}
...
```

This will inject `field_name=5` and `other_field='abc'` as fields of the flow and its components before starting its run, so the blocks would also have access to those fields. Note that you can also declare a `common` dict for blocks, but it's generally recommended to use default values for inputs instead.

- Using parametrize - you can specify fields for blocks or flows by calling their 'parametrize' or 'params' class method.

  For example:

```
class DemoFlow(TestFlow):
    blocks = (DemoBlock,
              DemoBlock.parametrize(field_name=5,
                                    other_field='abc'))
```

will create two blocks under the `DemoFlow`, one `DemoBlock` block with the default values for `field_name` and `other_field` (which can be set by defining them as class fields for the block for example, see optional inputs and fields section), and a second `DemoBlock` with `field_name=5` and `other_field='abc'` injected into the block instance (at runtime).

Regarding priorities hierarchy between the methods, it follows two rules:

1. For a single component, calling `parametrize` on it overrides the values set through `common`.
2. `common` and `parametrize` of sub-components are stronger than the values passed by containing hierarchies. E.g. `common` values of a flow are of lower priority than the `parametrize` values passed to the blocks under it.

Example

```
from rotest.core import TestBlock, BlockInput, BlockOutput
class DoSomethingBlock(TestBlock):
    """A block that does something.

    Attributes:
        resource1 (object): resource the block uses.
        input2 (object): input for the block.
        optional3 (object): optional input for the block.
    """
    mode = MODE_CRITICAL

    resource1 = BlockInput()
    input2 = BlockInput()
    optional3 = BlockInput(default=0)

    output1 = BlockOutput()

    def test_method(self):
        self.logger.info("Doing something")
        value = self.resource1.do_something(self.input2, self.optional3)
        self.output1 = value * 5  # This will be shared with siblings

...

class DemoFlow(TestFlow):
    resource1 = SomeResourceClass(some_limitation=LIMITATION)

    common = {'input2': INPUT_VALUE}

    blocks = (DemoBlock1,
              DemoBlock2,
              DemoBlock1,
              DoSomethingBlock.params(optional3=5),
              DoSomethingBlock,
              DemoBlock1.params(mode=MODE_FINALLY))
```

Sub-flows

A flow may contain not only test-block, but also test-flows under it. This feature can be used to wrap together blocks that tend to come together and also to create sub-procedures (if a test block is comparable to a simple function - it may have inputs and outputs and does a simple action, then a sub-flow can be considered a complex function, which invokes other simpler functions). Note that a sub-flow behaves exactly like a block, meaning, you can call parametrize on it, set a mode to it, it can't be filtered or skipped with delta, etc. This can give extra flexibility when composing flows with complex scenarios, for example:

```
Flow
|___BlockA
|___BlockB
|___BlockC
|___BlockD
```

If you want that block B will only run if block A passed, and that block D will only run if block C passed, but also to keep A and C not dependent, doing so is impossible without the usage of sub flows. But the scenario can be coded in the following manner:

```
Flow
|___SubFlow1 (mode optional)
    |___BlockA (mode critical)
    |___BlockB (mode critical)
|___SubFlow2 (mode optional)
    |___BlockC (mode critical)
    |___BlockD (mode critical)
```

Common *mistakes* when writing sub-flows:

- Flows can't declare inputs and outputs, only blocks can. They can, however, declare *mode* and *common* and be parametrized.
- Declared or imported sub-flows will be caught by the Rotest tests discoverer, than means that it will also try to run then separately. To avoid that, can either use –filter to run only specific flows or declare the sub-flows abstract using *__test__ = False*:

```
from rotest.core import TestFlow, create_flow, MODE_CRITICAL, MODE_OPTIONAL

class DemoSubFlow(TestFlow):
    __test__ = False

    mode = MODE_OPTIONAL

    blocks = (DemoBlock1,
              DemoBlock2,
              DemoBlock1)


class DemoFlow(TestFlow):
    resource1 = SomeResourceClass(some_limitation=LIMITATION)

    blocks = (DemoSubFlow,
              DemoSubFlow.params(input1=3),
              DemoSubFlow.params(mode=MODE_OPTIONAL))
```

Anonymous test-flows

Sub-flows can be created on-the-spot using the 'create_flow' function, to avoid defining classes. The functions gets the following arguments:

- `blocks` - list of the flow's components.
- `name` - name of the flow, default value is "AnonymousTestFlow", but it's recommended to override it.
- `mode` - mode of the new flow. Either `MODE_CRITICAL`, `MODE_OPTIONAL` or `MODE_FINALLY`. Default is `MODE_CRITICAL`.
- `common` - dict of initial fields and values for the new flow, same as the class variable 'common', default is empty dict.

```
from rotest.core.flow import TestFlow, create_flow

class DemoFlow(TestFlow):
    resource1 = SomeResourceClass(some_limitation=LIMITATION)

    blocks = (DemoBlock1,
```

(continues on next page)

(continued from previous page)

```
                DemoBlock2,
                DemoBlock1,
                create_flow(name="TestSomethingFlow",
                            common={"input2": "value1"}
                            mode=MODE_OPTIONAL,
                            blocks=[DoSomethingBlock,
                                    DoSomethingBlock.params(optional3=5)]),
                create_flow(name="TestAnotherThingFlow",
                            common={"input2": "value2"}
                            mode=MODE_OPTIONAL,
                            blocks=[DoSomethingBlock,
                                    DoSomethingBlock.params(optional3=5)]),
                DemoBlock1.params(mode=MODE_FINALLY))
```

Pipes

Since blocks are meant to be generic, sometimes the naming of their outputs and inputs won't align with other (more proprietary) blocks.

`Pipe` is the solution to this problem. With it, you can:

- Redirect values into blocks' inputs.
- Rename blocks' outputs.
- Adjust or transform values.

Consider the following code:

```
from rotest.core import TestBlock, TestFlow, BlockInput, BlockOutput


class DoSomethingBlock(TestBlock):
    output1 = BlockOutput()

    def test_method(self):
        self.output1 = 5

class ValidateSomethingBlock(TestBlock):
    input1 = BlockInput()

    def test_method(self):
        self.assertEqual(self.input1, 6)


class DemoFlow(TestFlow):
    blocks = (DoSomethingBlock,
              ValidateSomethingBlock)
```

The flow above can't run, since the blocks under *DemoFlow* don't connect properly - *ValidateSomethingBlock* doesn't get its required input.

But we can redirect *input1* to *output1* using `Pipe` in one of the following ways:

```
from rotest.core import TestFlow, Pipe
```

(continues on next page)

(continued from previous page)

```
class DemoFlow(TestFlow):
    blocks = (DoSomethingBlock.params(output1=Pipe('input1')),
              ValidateSomethingBlock)
```

```
from rotest.core import TestFlow, Pipe


class DemoFlow(TestFlow):
    blocks = (DoSomethingBlock,
              ValidateSomethingBlock.params(input1=Pipe('output1')))
```

```
from rotest.core import TestFlow, Pipe


class DemoFlow(TestFlow):
    common = {'input1': Pipe('output1')}

    blocks = (DoSomethingBlock,
              ValidateSomethingBlock)
```

```
from rotest.core import TestFlow, Pipe


class DemoFlow(TestFlow):
    common = {'output1': Pipe('input1')}

    blocks = (DoSomethingBlock,
              ValidateSomethingBlock)
```

Note that the use of `common` applies the pipe to all the blocks under the flow, and it overrides both *BlockInput* and *BlockOutput* instances with the given name.

Furthermore, we can manipulate values using `Pipe` (this can be done both to inputs and outputs):

```
from rotest.core import TestFlow, Pipe


class DemoFlow(TestFlow):
    blocks = (DoSomethingBlock.params(output1=Pipe('input1', formula=lambda x: x+1)),
              ValidateSomethingBlock)
```

In the example above, at the end of *DoSomethingBlock* two things would happen: * *output1* 's value will be transformed using the formula - from 5 to 6. * *output1* will change its name to *input1* before being shared.

2.3 Debugging

Rotest comes with easy ways to debug tests:

- Post run

  The builtin features in Rotest help you greatly when trying to figure out what went wrong in a test.

  - Logs of the tests can be found in the working directory.
  - `excel` output handler created a summary excel file in the working directory.

  - `artifact` output handler creates a zip of the working directory and sends it to the artifacts directory.
  - `save-state` command line option stores the state of resources into the working directory.
  - `remote` and `db` save the tests' metadata into the db, including traceback and timestamps, for future usage and research.

- Developing and real-time debugging
  - When running tests locally, using the *ipdbugger* (`--debug` flag) can be a real life saver. It pops an ipdb interactive shell whenever an unexpected exception occurs (including failures) without exiting the scope of the test, giving the user full control over it.

    For example, if an AttributeError has occurred, you can add the missing attribute via the interactive shell, then use *jump* or *retry* to re-run code segments. If your tests are based on Blocks and Flows methodology (see *Blocks code architecture*), you can use the *TestFlow* methods *list_blocks* and *jump_to* to control the flow of the test in the same way. E.g.

```
self.parent.list_blocks()  # Prints the hierarchy down from the parent flow
self.parent.jump_to(1)  # Jumps to the beginning of the block at index 1
```

  - It is also recommended to use `rotest shell` when debugging new code, especially when writing new *TestFlows* and *TestBlocks* (use the *shared_data* and *run_block* methods to simulate a containing TestFlow). Combining with IPython's `autoreload` ability, writing tests this way can be made easy and quick.

2.4 Adding Custom Output Handlers

2.4.1 Third Party Output Handlers

- rotest_reportportal
  - Plugin to the amazing Report Portal system, that enables viewing test results and investigating them.

2.4.2 How to Make Your Own Output Handler

You can make your own Output Handler, following the next two steps:

- Inheriting from *rotest.core.result.handlers.abstract_handler.AbstractResultHandler*, and overriding the relevant methods.
- Register the above inheriting class as an entrypoint in your setup.py file inside `setup()`:

```
entry_points={
    "rotest.result_handlers":
        ["<handler tag, e.g. my_handler> = <import path to the monitor's
↪module>:<monitor class name>"]
},
```

- Make sure it's being installed in the environment by calling

```
python setup.py develop
```

For an example, you can refer to rotest_reportportal plugin.

2.4.3 Available Events

The available methods of an output handler:

class rotest.core.result.handlers.abstract_handler.**AbstractResultHandler**(*main_test=None*, **args*, ***kwargs*)

Result handler interface.

Defines the required interface for all the result handlers.

main_test
the main test instance (e.g. TestSuite instance or TestFlow instance).

Type rotest.core.abstract_test.AbstractTest

add_error(*test*, *exception_string*)
Called when an error has occurred.

Parameters

- **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.
- **exception_string** (*str*) – exception description.

add_expected_failure(*test*, *exception_string*)
Called when an expected failure/error occurred.

Parameters

- **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.
- **exception_string** (*str*) – exception description.

add_failure(*test*, *exception_string*)
Called when an error has occurred.

Parameters

- **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.
- **exception_string** (*str*) – exception description.

add_info(*test*, *msg*)
Called when a test registers a success message.

Parameters

- **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.
- **msg** (*str*) – success message.

add_skip(*test*, *reason*)
Called when a test is skipped.

Parameters

- **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.
- **reason** (*str*) – reason for skipping the test.

add_success(*test*)
Called when a test has completed successfully.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

add_unexpected_success(*test*)
Called when a test was expected to fail, but succeed.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

print_errors(*tests_run*, *errors*, *skipped*, *failures*, *expected_failures*, *unexpected_successes*)
Called by TestRunner after test run.

Parameters

- **tests_run** (*number*) – count of tests that has been run.
- **errors** (*list*) – error tests details list.
- **skipped** (*list*) – skipped tests details list.
- **failures** (*list*) – failed tests details list.
- **expected_failures** (*list*) – expected-to-fail tests details list.
- **unexpected_successes** (*list*) – unexpected successes tests details list.

setup_finished(*test*)
Called when the given test finished setting up.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

should_skip(*test*)
Check if the test should be skipped.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

Returns skip reason if the test should be skipped, None otherwise.

Return type str

start_composite(*test*)
Called when the given TestSuite is about to be run.

Parameters **test** (*rotest.core.suite.TestSuite*) – test item instance.

start_teardown(*test*)
Called when the given test is starting its teardown.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

start_test(*test*)
Called when the given test is about to be run.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

start_test_run()
Called once before any tests are executed.

stop_composite(*test*)
Called when the given TestSuite has been run.

Parameters **test** (*rotest.core.suite.TestSuite*) – test item instance.

stop_test(*test*)
Called when the given test has been run.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

stop_test_run()
Called once after all tests are executed.

update_resources(*test*)
Called once after locking the tests resources.

Parameters **test** (*rotest.core.abstract_test.AbstractTest*) – test item instance.

2.5 Test Monitors

2.5.1 Purpose

Monitors are custom output handlers that are meant to give further validation of tests in runtime, or save extra information about the tests.

The features of monitors:

- They can be applied or dropped as easily as adding a new output handler to the list.
- They enable extending sets of tests with additional validations without altering their code.
- They can run in the background (in another thread).

A classic example is monitoring CPU usage during tests, or a resource's log file.

2.5.2 Writing A Monitor

There are two monitor classes which you can inherit from:

class rotest.core.result.monitor.monitor.**AbstractMonitor**(**args*, ***kwargs*)
Abstract monitor class.

CYCLE
sleep time in seconds between monitor runs.

Type number

SINGLE_FAILURE
whether to continue running the monitor after it had failed or not.

Type bool

Note: When running in multiprocess, regular output handlers will be used by the main process, and the monitors will be run by each worker, since they use tests' attributes (resources, for example) that aren't available in the main process.

---

fail_test(*test*, *message*)
Add a monitor failure to the test without stopping it.

Parameters

- **test** (*object*) – test item instance.
- **message** (*str*) – failure message.

run_monitor(*test*)
The monitor main procedure.

class rotest.core.result.monitor.monitor.**AbstractResourceMonitor**(**args*, ***kwargs*)

Abstract cyclic monitor that depends on a resource to run.

This class extends the AbstractMonitor behavior and also waits for the resource to be ready for work before calling *run_monitor*.

RESOURCE_NAME
expected field name of the resource in the test.

Type str

There are two types of monitors:

- Monitors that only react to test events, e.g. taking a screen-shot on error.

  Since monitors inherit from `AbstractResultHandler`, you can react to any test event by overriding the appropriate method.

  See *Available Events* for a list of events.

  Each of those event methods gets the test instance as the first parameter, through which you can access its fields (`test.<resource>`, `test.config`, `test.work_dir`, etc.)

- Monitors that run in the background and periodically save data or run a validation, like the above suggested CPU usage monitor.

  To create such a monitor, simply override the class field `CYCLE` and the method `run_monitor`.

  Again, the `run_monitor` method (which is called periodically after *setUp* and until *tearDown*) gets the test instance as a parameter, through which you can get what you need.

  Note that the monitor thread is created only for upper tests, i.e. `TestCases` or topmost `TestFlows`.

  Remember that you might need to use some synchronization mechanism since you're running in a different thread yet using the test's own resources.

Use the method `fail_test` to add monitor failures to your tests in the background, e.g.

```
self.fail_test(test, "Reached 100% CPU usage")
```

Note that when using `TestBlocks` and `TestFlows`, you might want to limit your monitor events to only be applied on main tests and not sub-components (`run_monitor` already behaves that way by default). For your convenience, you can use the following decorators on the overridden event methods to limit their activity:

rotest.core.result.monitor.monitor.**skip_if_case**(*func*)
Avoid running the decorated method if the test is a TestCase.

rotest.core.result.monitor.monitor.**skip_if_flow**(*func*)
Avoid running the decorated method if the test is a TestFlow.

rotest.core.result.monitor.monitor.**skip_if_block**(*func*)
Avoid running the decorated method if the test is a TestBlock.

rotest.core.result.monitor.monitor.**skip_if_not_main**(*func*)
Avoid running the method if the test is a TestBlock or sub-flow.

rotest.core.result.monitor.monitor.**require_attr**(*resource_type*)
Avoid running the decorated method if the test lacks an attribute.

Parameters **resource_type** (*str*) – name of the attribute to search.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

E

F

M

P

R

S

U