Rotest Documentation

Release 2.4.1

gregoil

Apr 12, 2018

Contents

1 Basic **1**
- 1.1 Rotest . . . 1
- 1.2 Getting Started . . . 3
- 1.3 Command Line Options . . . 10
- 1.4 Output Handlers . . . 18
- 1.5 Configurations . . . 20

2 Advanced **23**
- 2.1 Adding Custom Output Handlers . . . 23
- 2.2 Blocks code architecture . . . 25

3 Indices and tables **31**

CHAPTER 1

Basic

1.1 Rotest

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.core.runner import main
from rotest.core.case import TestCase


class SimpleCalculationTest(TestCase):
    calculator = Calculator()

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


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

The test can 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:

```
$ python test.py --help
Usage: test.py [options]

Options:
  -h, --help            show this help message and exit
  -c CONFIG_PATH, --config-path=CONFIG_PATH
                        Tests' configuration file path
```

```
  -s, --save-state      Enable save state
  -d DELTA_ITERATIONS, --delta-iterations=DELTA_ITERATIONS
                        Enable run of unsuccessful tests only, enter the
                        number of times the failed tests should run
  -p PROCESSES, --processes=PROCESSES
                        Use multiprocess test runner
  -o OUTPUTS, --outputs=OUTPUTS
                        Output handlers separated by comma. Options: dots,
                        xml, full, remote, tree, excel, db, artifact,
                        signature, loginfo, logdebug, pretty
  -f FILTER, --filter=FILTER
                        Run only tests that match the filter expression, e.g
                        "Tag1* and not Tag13"
  -n RUN_NAME, --name=RUN_NAME
                        Assign run name
  -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 RESOURCES, --resources=RESOURCES
                        Specific resources to request by name
```

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.case.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.case import TestCase


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

- `rotest.core.suite.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.suite.TestSuite` can hold each of the following:
  - `rotest.core.case.TestCase` classes.
  - `rotest.core.suite.TestSuite` classes.
  - The more complex concept of `rotest.core.flow.TestFlow` classes.

```
from rotest.core.suite 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
```

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

```
rotest:
    host: localhost
    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 *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.core.runner import main
from rotest.core.case import TestCase


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


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

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'
```

1.2.3 Using 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.
- How to write tests that use resources.

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 import resource_data


class CalculatorData(resource_data.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

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

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

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

Note the following:

- There is a use in 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()`.

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 --run-django-server --django-port 8080 --daemon
Running in detached mode (as daemon)
```

Warning: The *`--daemon`* option is not implemented in Windows.

A few explanations about this command:

- If given the *`--run-django-server`* option, it runs the Django admin panel as well. We'll access it in the next section.
- If given the *`--django-port`* option, it uses this value as the port of the Django admin panel. If not given, it defaults to `8000`.
- If given the *`--daemon`* or *`-D`* option, the program runs in the background.

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://127.0.0.1:8080/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`:

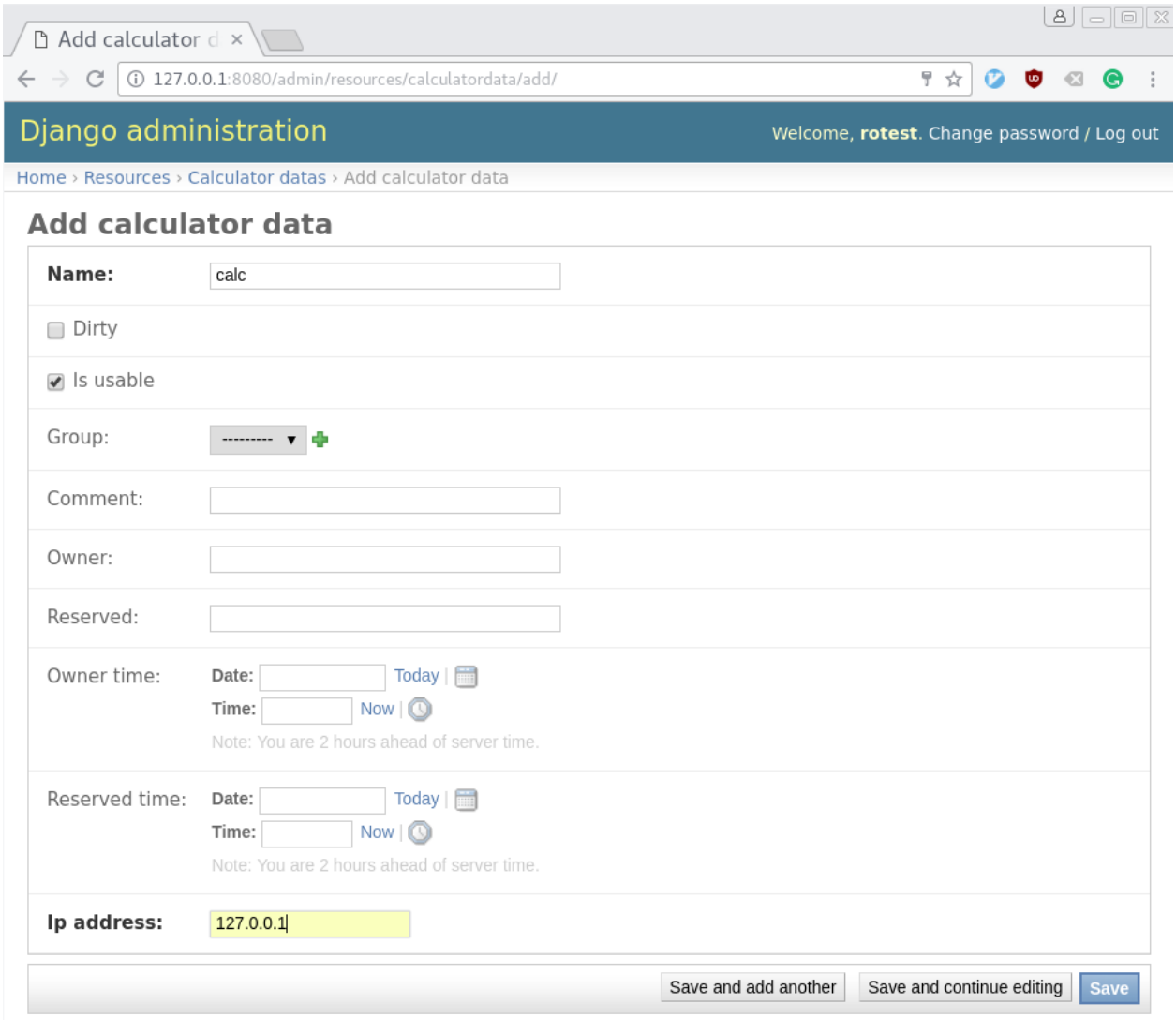


Fig. 1.1: Adding a resource via Dango admin

Writing a Resource-Based Test

In this section, we are going to add a resource request 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.runner import main
from rotest.core.case import TestCase

from resources.calculator import Calculator


class AddTest(TestCase):
    calc = Calculator()

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


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

Now, let's run the test:

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

Ran 1 test in 0.160s

OK
```

Well done! You've just written your first resource oriented test, that asserts the behaviour of a simple addition of a Calculator resource.

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`**.

Getting Help

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

The *`--help`* option is here to help:

```
$ rotest-server --help
Run resource manager server.

Usage:
```

```
    rotest-server [--server-port <port>] [--run-django-server]
                  [--django-port <port>] [-D | --daemon]

Options:
    -h --help
        show this help message and exit

    --server-port <port>
        port for communicating with the client

    --run-django-server
        run the Django frontend as well

    --django-port <port>
        set Django's port [default: 8000]

    -D --daemon
        run as a daemon
```

Selecting Server's Port

--server-port <port>
Select the port for communicating with the client.

By default, the server uses the specified configuration for the port (see *ROTEST_SERVER_PORT*), or defaults to 7777. If this port is already in use and you'd like to change it, use option *--server-port*:

```
$ rotest-server --server-port 8888
Running in attached mode
<2018-01-24 18:49:19,654>[DEBUG][main@98]: Starting resource manager, port:8888
<2018-01-24 18:49:19,655>[DEBUG][manager@101]: Resource manager main thread started
```

Running Django's Frontend

--run-django-server
Run the Django frontend as well.

--django-port <port>
Set Django's port (defaults to 8000).

As well as the server, one may want to run the Django's server, which enables editing and viewing the database that contains the resources. Use option *--run-django-server* to run the Django's server, and optionally option *--django-port* to choose the used port. It defaults to port 8000:

```
$ rotest-server --run-django-server --django-port 9999
Running in attached mode
Running the Django server as well
<2018-01-24 18:54:46,590>[DEBUG][main@98]: Starting resource manager, port:7778
<2018-01-24 18:54:46,591>[DEBUG][manager@101]: Resource manager main thread started
Performing system checks...

System check identified no issues (0 silenced).
January 24, 2018 - 18:54:47
Django version 1.7.11, using settings 'rotest_template.settings'
Starting development server at http://0.0.0.0:9999/
Quit the server with CONTROL-C.
```

Daemon Mode

-D, --daemon
Run as a daemon process.

> **Warning:** Not implemented in Windows.

A common case is to run the server in the background. Use options *--daemon* or *-D* to run the server as a daemon process:

```
$ rotest-server --daemon
Running in detached mode (as daemon)
```

You can combine it with the other options, like *--run-django-server*.

1.3.2 Client Options

Getting Help

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

First, and most important, using the help options *-h* or *--help*:

```
$ python some_test_file.py -h
Usage: some_test_file.py [options]

Options:
  -h, --help            show this help message and exit
  -c CONFIG_PATH, --config-path=CONFIG_PATH
                        Tests' configuration file path
  -s, --save-state      Enable save state
  -d DELTA_ITERATIONS, --delta-iterations=DELTA_ITERATIONS
                        Enable run of failed tests only, enter the number of
                        times the failed tests should run
  -p PROCESSES, --processes=PROCESSES
                        Use multiprocess test runner
  -o OUTPUTS, --outputs=OUTPUTS
                        Output handlers separated by comma. Options: dots,
                        xml, full, remote, tree, excel, db, artifact,
                        signature, loginfo, logdebug, pretty
  -f FILTER, --filter=FILTER
                        Run only tests that match the filter expression, e.g
                        "Tag1* and not Tag13"
  -n RUN_NAME, --name=RUN_NAME
                        Assign run name
  -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 RESOURCES, --resources=RESOURCES
                        Specific resources to request by name
```

Listing and Filtering

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

-f FILTER, **--filter** FILTER
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:

```
$ python 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:

```
$ python some_test_file.py -f FLOW -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

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):

```
    $ python 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
```

Stopping at first failure

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

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

```
$ python 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.

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

```
$ python 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)
   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.

Retrying Tests

-d DELTA_ITERATIONS,
--delta DELTA_ITERATIONS
--delta-iterations 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-iterations* or *-d*:

```
$ python some_test_file.py --delta-iterations 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
----------------------------------------------------------------------
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:

```
$ python 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()

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

You can request resources by their names:

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

Alternatively, you can make more complex queries:

```
$ python 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:

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

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

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
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 XML & 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. Alternatively, it can output a Junit-compatible XML, which lots of reporting systems can parse and display. The two relevant options are `-o excel` and `-o xml`.

Those artifacts are 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*.

1.5 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 on the server's side, to be used for communication with clients.

To define the relevant server's port the 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 `7777`.

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_fir` in the configuration file:

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

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

CHAPTER 2

Advanced

2.1 Adding Custom Output Handlers

2.1.1 Third Party Output Handlers

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

2.1.2 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 a setup.py file, and make sure it's being install on the environment.

For an example, please refer to rotest_reportportal plugin.

2.1.3 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
rotest.core.abstract_test.AbstractTest – the main test instance (e.g. TestSuite instance or TestFlow instance).

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_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.

2.2 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 a static list or tuple in the new block's class of fields the block needs to run. For example, defining in the block's scope

```
class DemoBlock(TestBlock):
    inputs = ('field_name', 'other_field')
...
```

   will validate that the block instance will have all those field before running the parent flow. The inputs validation (which happens before running the topmost flow) passes if those fields are present in the block (e.g. the fields were set using parametrize), or if a previous sibling component will share those fields in runtime.

2. `outputs`: define a static list or tuple in the new block's class of fields the block would share in its run. For example, defining in the block's scope

```
class DemoBlock(TestBlock):
    outputs = ('field_name', 'other_field')
...
```

   means declaring that the block would calculate and share (using the `share_data` method) those fields, so that components following the block would get those fields at runtime. 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. `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 saves values to be passed to it when it will be run.
3. `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`)
   (a) `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.
   (b) `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).
   (c) `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.
4. `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.
- Sharing data - if one block writes somewhere in its test method:

  ```
  self.share_data(field_name=value)
  ```

  then all the components under the parent flow are injected (into their instance - self) where the field `field_name` is with value `value`.

- Setting initial data to the test flow - you can set initial data to the components of flows 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. This is the same as sharing those fields at the beginning of the flow's setUp method, using `share_data()`.

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

Example

```
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
    inputs = ('resource1', 'input2')

    optional3 = 0

    def test_method(self):
        """Do something."""
        self.logger.info("Doing something")
        self.resource1.do_something(self.input2, self.optional3)

...

class DemoFlow(TestFlow):
    """Demo test-flow."""
    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)
```

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):
    """Demo test-flow."""
    resource1 = SomeResourceClass(some_limitation=LIMITATION)

    blocks = (DemoBlock1,
              DemoBlock2,
              DemoBlock1,
              create_flow(name="TestSomethingFlow",
```

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

Optional inputs and fields

Mainly for convenience purposes, we sometimes want to have default values for fields of blocks (inputs), just like we want default values for functions' arguments. Doing so is possible using the fact that passing inputs to blocks is done by injecting fields into their instance. For example:

```
class DemoBlock(TestBlock):
    """Demo block.

    Attributes:
        argument1 (number): block's first argument.
        argument2 (number): block's second argument.
        argument3 (number): block's third argument.
    """
    mode = MODE_CRITICAL
    inputs = ('argument1', 'argument2', 'argument3')

    argument2 = 0  # Setting default value to 0
    argument3 = 1  # Setting default value to 1

    def test_method(self):
        ...
```

Defining the block so is equivalent to defining the following function:

```
def DemoBlock(argument1, argument2=0, argument3=1):
    ...
```

Doing so, means that you wouldn't have to pass values to the block for the parameters 'argument2' and 'argument3' (on ways of passing values to block's parameters, see the *Sharing data* section), meaning that all the following instantiations wouldn't raise an error due to input validation:

```
DemoBlock.params(argument1=5)  # arguments = 5,0,1
DemoBlock.params(argument1=5,argument2=3)  # arguments = 5,3,1
DemoBlock.params(argument1=5,argument3=4)  # arguments = 5,0,4
DemoBlock.params(argument1=5,argument2=3,argument3=6)  # arguments = 5,3,6
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

Symbols

–delta DELTA_ITERATIONS
    rotest command line option, 16
–delta-iterations DELTA_ITERATIONS
    rotest command line option, 16
–django-port <port>
    rotest-server command line option, 11
–run-django-server
    rotest-server command line option, 11
–server-port <port>
    rotest-server command line option, 11
-D, –daemon
    rotest-server command line option, 12
-D, –debug
    rotest command line option, 15
-F, –failfast
    rotest command line option, 14
-d DELTA_ITERATIONS,
    rotest command line option, 16
-f FILTER, –filter FILTER
    rotest command line option, 13
-h, –help
    rotest command line option, 12
    rotest-server command line option, 10
-l, –list
    rotest command line option, 13
-o OUTPUTS, –outputs OUTPUTS
    rotest command line option, 17
-p PROCESSES, –processes PROCESSES
    rotest command line option, 17
-r <query>, –resources <query>
    rotest command line option, 17

A

AbstractResultHandler (class in rotest.core.result.handlers.abstract_handler), 23
add_error() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 24
add_expected_failure() (rotest.core.result.handlers.abstract_handler.Abstrac method), 24
add_failure() (rotest.core.result.handlers.abstract_handler.AbstractResultHa method), 24
add_skip() (rotest.core.result.handlers.abstract_handler.AbstractResultHand method), 24
add_success() (rotest.core.result.handlers.abstract_handler.AbstractResultH method), 24
add_unexpected_success() (rotest.core.result.handlers.abstract_handler.AbstractResultHandl method), 24
ARTIFACTS_DIR, 22

D

DJANGO_SETTINGS_MODULE, 22

E

environment variable
    ARTIFACTS_DIR, 22
    DJANGO_SETTINGS_MODULE, 22
    ROTEST_HOST, 21
    ROTEST_RESOURCE_REQUEST_TIMEOUT, 21, 22
    ROTEST_SERVER_PORT, 11, 21
    ROTEST_WORK_DIR, 21

M

main_test (rotest.core.result.handlers.abstract_handler.AbstractResultHandl attribute), 23

P

print_errors() (rotest.core.result.handlers.abstract_handler.AbstractResultHa method), 24

R

rotest command line option
    –delta DELTA_ITERATIONS, 16
    –delta-iterations DELTA_ITERATIONS, 16
    -D, –debug, 15

-F, –failfast, 14
-d DELTA_ITERATIONS,, 16
-f FILTER, –filter FILTER, 13
-h, –help, 12
-l, –list, 13
-o OUTPUTS, –outputs OUTPUTS, 17
-p PROCESSES, –processes PROCESSES, 17
-r <query>, –resources <query>, 17
rotest-server command line option
–django-port <port>, 11
–run-django-server, 11
–server-port <port>, 11
-D, –daemon, 12
-h, –help, 10
ROTEST_HOST, 21
ROTEST_RESOURCE_REQUEST_TIMEOUT, 22
ROTEST_SERVER_PORT, 11, 21
ROTEST_WORK_DIR, 21

S

setup_finished() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 24
should_skip() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
start_composite() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
start_teardown() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
start_test() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
start_test_run() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
stop_composite() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
stop_test() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25
stop_test_run() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25

U

update_resources() (rotest.core.result.handlers.abstract_handler.AbstractResultHandler method), 25