Seaworthy Documentation

Release 0.4.2

Jamie Hewland & Jeremy Thurgood

Jan 30, 2019

Contents:

Seaworthy is a test harness for Docker container images. It allows you to use Docker containers and other Docker resources as fixtures for tests written in Python.

Seaworthy supports Python 3.4 and newer. You can find more information in the documentation.

A demo repository is available with a set of Seaworthy tests for a simple Django application. Seaworthy is also introduced in our blog post on continuous integration with Docker on Travis CI.

For more background on the design and purpose of Seaworthy, see our PyConZA 2018 talk (slides).

CHAPTER 1

Quick demo

First install Seaworthy along with pytest using pip:

```
pip install seaworthy[pytest]
```

Write some tests in a file, for example, `test_echo_container.py`:

```
from seaworthy.definitions import ContainerDefinition

container = ContainerDefinition(
    'echo', 'jmalloc/echo-server',
    wait_patterns=[r'Echo server listening on port 8080'],
    create_kwargs={'ports': {'8080': None}})
fixture = container.pytest_fixture('echo_container')


def test_echo(echo_container):
    r = echo_container.http_client().get('/foo')
    assert r.status_code == 200
    assert 'HTTP/1.1 GET /foo' in r.text
```

Run pytest:

```
pytest -v test_echo_container.py
```

CHAPTER 2

Project status

Seaworthy should be considered alpha-level software. It is well-tested and works well for the first few things we have used it for, but we would like to use it for more of our Docker projects, which may require some parts of Seaworthy to evolve further. See the project issues for known issues/shortcomings.

The project was originally split out of the tests we wrote for our docker-django-bootstrap project. There are examples of Seaworthy in use there.

2.1 Getting started

2.1.1 Installation

Seaworthy can be installed using pip:

```
pip install seaworthy
```

The pytest and testtools integrations can be used if those libraries are installed, which can be done using extra requirements:

```
pip install seaworthy[pytest,testtools]
```

2.1.2 Defining containers for tests

Containers should be defined using subclasses of *ContainerDefinition*. For example:

```
from seaworthy.definitions import ContainerDefinition
from seaworthy.logs import output_lines


class CakeContainer(ContainerDefinition):
    IMAGE = 'acme-corp/cake-service:chocolate'
```

(continues on next page)

(continued from previous page)

```
    WAIT_PATTERNS = (
        r'cake \w+ is baked',
        r'cake \w+ is served',
    )

    def __init__(self, name):
        super().__init__(name, self.IMAGE, self.WAIT_PATTERNS)

    # Utility methods can be added to the class to extend functionality
    def exec_cake(self, *params):
        return output_lines(self.inner().exec_run(['cake'] + params))
```

`WAIT_PATTERNS` is a list of regex patterns. Once these patterns have been seen in the container logs, the container is considered to have started and be ready for use. For more advanced readiness checks, the *`wait_for_start()`* method should be overridden.

This container can then be used as fixtures for tests in a number of ways, the easiest of which is with pytest:

```
import pytest

container = CakeContainer('test')
fixture = container.pytest_fixture('cake_container')

def test_type(cake_container):
    output = cake_container.exec_cake('type')
    assert output == ['chocolate']
```

A few things to note here:

- The *`pytest_fixture()`* method returns a pytest fixture that ensures that the container is created and started before the test begins and that the container is stopped and removed after the test ends.
- The scope of the fixture is important. By default, pytest fixtures have function scope, which means that for each test function the fixture is completely reinitialized. Creating and starting up a container can be a little slow, so you need to think carefully about what scope to use for your fixtures. See *`ContainerDefinition.clean`* for a way to avoid container setup/teardown overhead.

For simple cases, *`ContainerDefinition`* can be used directly, without subclassing:

```
container = ContainerDefinition(
    'test', 'acme-corp/soda-service:cola', [r'soda \w+ is fizzing'])
fixture = container.pytest_fixture('soda_container')

def test_refreshment(soda_container):
    assert 'Papor-Colla Corp' in soda_container.get_logs()
```

Note that pytest is not required to use Seaworthy and there are several other ways to use the container as a fixture. For more information see *Test framework integration* and *Resource definitions & helpers*.

2.2 Resource definitions & helpers

Two important abstractions in Seaworthy are resource *definitions* and *helpers*. These provide test-oriented interfaces to all of the basic (non-Swarm) Docker resource types.

2.2.1 Definitions

Resource definitions provide three main functions:

- Make it possible to *define* resources before those resources are actually created in Docker. This is important for creating test fixtures—if we can define everything about a resource before it is created, then we can create the resource when it is needed as a fixture for a test.
- Simplify the setup and teardown of resources before and after tests. For example, *ContainerDefinition* can be used to check that a container has produced certain log lines before it is used in a test.
- Provide useful functionality to interact with and introspect resources. For example, the *http_client()* method can be used to get a simple HTTP client to make requests against a container.

Resource defintions can either be instantiated directly or subclassed to provide more specialised functionality.

For a simple volume, one could create an instance of *VolumeDefinition*:

```
from seaworthy.definitions import VolumeDefinition
from seaworthy.helpers import DockerHelper


docker_helper = DockerHelper()
volume = VolumeDefinition('persist', helper=docker_helper)
```

Using definitions in tests

Definitions can be used as fixtures for tests in a number of different ways.

As a context manager:

```
with VolumeDefinition('files', helper=docker_helper) as volume:
    assert volume.created

assert not volume.created
```

With the `as_fixture` decorator:

```
network = NetworkDefinition('lan_network', helper=docker_helper)

@network.as_fixture()
def test_network(lan_network):
    assert lan_network.created
```

When using pytest, it's easy to create a fixture:

```
container = ContainerDefinition('nginx', 'nginx:alpine')
fixture = container.pytest_fixture('nginx_container')

def test_nginx(nginx_container):
    assert nginx_container.created
```

You can also use classic xunit-style setup/teardown:

```
import unittest


class EchoContainerTest(unittest.TestCase):
```

(continues on next page)

(continued from previous page)

```
    def setUp(self):
        self.helper = DockerHelper()
        self.container = ContainerDefinition('echo', 'jmalloc/echo-server')
        self.container.setup(helper=self.helper)
        self.addCleanup(self.container.teardown)

    def test_container(self):
        self.assertTrue(self.container.created)
```

Relationship to helpers

Every resource definition instance needs to have a "helper" set before it is possible to actually create the Docker resource that the instance defines. Resource helpers are described in more detail later in this section, but for now, know that a helper needs to be provided to the definition in one of three ways:

1. Using the `helper` keyword argument in the constuctor:

   ```
   helper = DockerHelper()
   network = NetworkDefinition('net01', helper=helper)
   network.setup()
   ```

2. Using the `helper` keyword argument in the `setup()` method:

   ```
   helper = DockerHelper()
   volume = VolumeDefinition('vol02')
   volume.setup(helper=helper)
   ```

3. Directly, using the `set_helper()` method:

   ```
   helper = DockerHelper()
   container = ContainerDefinition('con03', 'nginx:alpine')
   container.set_helper(helper)
   container.setup()
   ```

This only needs to be done once for the lifetime of the definition.

For the most part, interaction with Docker should almost entirely occur via the definitions, but the definition objects need the helpers to actually interact with Docker.

Mapping to Docker SDK types

Each resource definition wraps a model from the Docker SDK for Python. The underlying model can be accessed via the `inner()` method, after the resource has been created. The mapping is as follows:

Seaworthy resource definition	Docker SDK model
ContainerDefinition	`docker.models.containers.Container`
NetworkDefinition	`docker.models.networks.Network`
VolumeDefinition	`docker.models.volumes.Volume`

2.2.2 Helpers

Resource helpers provide two main functions:

- Namespacing of resources: by prefixing resource names, the resources are isolated from other Docker resources already present on the system.
- Teardown (cleanup) of resources: when the tests end, the networks, volumes, and containers used in those tests are removed.

In addition, some of the behaviour around resource creation and removal is changed from the Docker defaults to be a better fit for a testing environment.

Accessing the various helpers is most easily done via the *DockerHelper*:

```
from seaworthy.helpers import DockerHelper


# Create a DockerHelper with the default namespace, 'test'
docker_helper = DockerHelper()

# Create a network using the NetworkHelper
network = docker_helper.networks.create('private')

# Create a volume using the VolumeHelper
volume = docker_helper.volumes.create('shared')

# Fetch (pull) an image using the ImageHelper
image = docker_helper.images.fetch('busybox')

# Create a container using the ContainerHelper
container = docker_helper.containers.create(
    'conny', image, network=network, volumes={volume: '/vol'})
```

The DockerHelper can be configured with a custom Docker API client. The default client can be configured using environment variables. See `docker.client.from_env()`.

Mapping to Docker SDK types

Each resource helper wraps a "model collection" from the Docker SDK. The underlying collection can be accessed via the `collection` attribute. The mapping is as follows:

Seaworthy resource helper	Docker SDK model collection
ContainerHelper	`docker.models.containers.ContainerCollection`
ImageHelper	`docker.models.images.ImageCollection`
NetworkHelper	`docker.models.networks.NetworkCollection`
VolumeHelper	`docker.models.volumes.VolumeCollection`

2.3 Test framework integration

We have strong opinions about the testing tools we use, and we understand that other people may have equally strong opinions that differ from ours. For this reason, we have decided that none of Seaworthy's core functionality will depend on pytest, testtools, or anything else that might get in the way of how people might wants to write their tests. On the other hand, we don't want to reinvent a bunch of integration and helper code for all the third-party testing tools we like, so we also provide optional integration modules where it makes sense to do so.

2.3.1 pytest

Seaworthy is a pytest plugin and all the functions and fixtures in the *seaworthy.pytest* module will be available when Seaworthy is used with pytest.

`docker_helper` fixture

A fixture for a *DockerHelper* instance is defined by default.

This fixture creates `DockerHelper` instances with default parameters and has **module-level scope**. Since all other Docker resource fixtures typically depend on the `docker_helper` fixture, resources must have a scope smaller than or equal to the `docker_helper`'s scope.

The defaults for this fixture can be overridden by defining a new `docker_helper` fixture using the *docker_helper_fixture()* fixture factory. For example:

```
from seaworthy.pytest.fixtures import docker_helper_fixture

docker_helper = docker_helper_fixture(scope='session', namespace='seaworthy')
```

... would change the scope of the `docker_helper` fixture to the session-level and change the namespace of created Docker resources to `seaworthy`.

`dockertest` decorator

The *dockertest()* decorator can be used to mark tests that *require* Docker to run. These tests will be skipped if Docker is not available. It's possible that some tests in your test suite may not require Docker and you may want to still be able to run your tests in an environment that does not have Docker available. The decorator can be used as follows:

```
@dockertest()
def test_docker_thing(cake_container):
    assert cake_container.exec_cake('variant') == ['gateau']
```

Fixture factories

A few functions are provided in the *seaworthy.pytest.fixtures* module that are factories for fixtures. The most important two are:

resource_fixture(*definition*, *name*, *scope='function'*, *dependencies=()*)

Create a fixture for a resource.

Note: This function returns a fixture function. It is important to keep a reference to the returned function within the scope of the tests that use the fixture.

```
fixture = resource_fixture(PostgreSQLContainer(), 'postgresql')

def test_container(postgresql):
    """Test something about the PostgreSQL container..."""
```

Parameters

- **definition** – A resource definition, one of those defined in the *seaworthy.definitions* module.
- **name** – The fixture name.
- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

Returns The fixture function.

clean_container_fixtures(*container*, *name*, *scope='class'*, *dependencies=()*)

Creates a fixture for a container that can be "cleaned". When a code block is marked with `@pytest.mark.clean_<fixture name>` then the `clean` method will be called on the container object before it is passed as an argument to the test function.

Note: This function returns two fixture functions. It is important to keep references to the returned functions within the scope of the tests that use the fixtures.

```
f1, f2 = clean_container_fixtures(PostgreSQLContainer(), 'postgresql')

class TestPostgresqlContainer
    @pytest.mark.clean_postgresql
    def test_clean_container(self, web_container, postgresql):
        """
        Test something about the container that requires it to have a
        clean state (e.g. database table creation).
        """

    def test_dirty_container(self, web_container, postgresql):
        """
        Test something about the container that doesn't require it to
        have a clean state (e.g. testing something about a dependent
        container).
        """
```

Parameters

- **container** – A "container" object that is a subclass of *ContainerDefinition*.
- **name** – The fixture name.
- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

Returns A tuple of two fixture functions.

2.3.2 testtools

We primarily use testtools when matching against complex data structures and don't use any of its test runner functionality. Currently, testtools matchers are only used for matching *PsTree* objects. See the API documentation for the *seaworthy.ps* module.

2.3.3 Testing our integrations

To make sure that none of the optional dependencies accidentally creep into the core modules (or other optional modules), we have several sets of tests that run in different environments:

- `tests-core`: This is a set of core tests that cover basic functionality. `tox -e py36-core` will run just these tests in an environment without any optional or extra dependencies installed.
- `tests-pytest`, etc.: These are tests for the optional pytest integration modules. `tox -e py36-testtools` will run just the `seaworthy.pytest` modules' tests in an environment with only the necessary dependencies installed.
- `tests-testtools`, etc.: These are tests for the optional testtools integration module. `tox -e py36-testtools` will run just the `seaworthy.testtools` module's tests.
- `tests`: These are general tests that are hard or annoying to write with only the minimal dependencies, so we don't have any tooling restrictions here. `tox -e py36-full` will run these, as well as all the other test sets mentioned above, in an environment with all optional dependencies (and potentially some additional test-only dependencies) installed.

2.4 Frequently asked questions

2.4.1 What about TestContainers?

Seaworthy's goals have some overlap with TestContainers, but our current primary use case is testing the behaviour of Docker images, rather than providing a way to use Docker containers to test other software. Also, Seaworthy is written in Python rather than Java.

2.4.2 What are the similarities between Seaworthy and docker-compose?

Seaworthy does try to reuse some of the default behaviour that docker-compose implements in order to make it easier and faster to start running containers.

- All Docker resources (networks, volumes, containers) are namespaced by prefixing the resource name, e.g. a container called `cake-service` could be namespaced to have the name `test_cake-service`.
- A new bridge network is created by default for containers where no network is specified.
- Containers are given network aliases with their names, making it easier to connect one container to another.

Both Seaworthy and docker-compose are built using the official Docker SDK for Python.

2.4.3 . . . what are the differences?

Seaworthy is fundamentally designed for a different purpose. docker-compose uses YAML files to define Docker resources—it does not have an API for this. With Seaworthy, all Docker resources are created *programmatically*, typically as fixtures for tests.

Seaworthy includes functionality specific to its purpose:

- Predictable setup/teardown processes for all resources.
- Various utilities for inspecting running containers, e.g. for matching log output, for listing running processes, or for making HTTP requests against containers.
- Integrations with popular Python testing libraries (pytest and testtools).

Seaworthy currently lacks some of the functionality of docker-compose:

- The ability to build images for containers
- Any sort of Docker Swarm functionality
- Any concept of multiple instances of containers
- Probably other things...

2.4.4 What about building images?

Seaworthy doesn't currently implement an interface for building images. In most cases, we expect users to build their images in a previous step of their continuous integration process and then use Seaworthy to test that image. However, there may be cases where having Seaworthy build Docker images would make sense, such as if an image is built purely to be used in tests.

2.5 API Reference

seaworthy	seaworthy
seaworthy.checks	Checks and test decorators for skipping tests that require Docker to be present.
seaworthy.client	A requests-based HTTP client for interacting with containers that have forwarded ports.
seaworthy.containers.nginx	
seaworthy.containers.postgresql	PostgreSQL container definition.
seaworthy.containers.rabbitmq	RabbitMQ container definition.
seaworthy.containers.redis	Redis container definition.
seaworthy.definitions	Wrappers over Docker resource types to aid in setup/teardown of and interaction with Docker resources.
seaworthy.helpers	Classes that track resource creation and removal to ensure that all resources are namespaced and cleaned up after use.
seaworthy.ps	Tools for asserting on processes running in containers using `ps`.
seaworthy.pytest	Some (optional) utilities for use with pytest.
seaworthy.pytest.checks	pytest mark to skip tests that require Docker.
seaworthy.pytest.fixtures	A number of pytest fixtures or factories for fixtures.
seaworthy.stream	
seaworthy.stream.logs	
seaworthy.stream.matchers	
seaworthy.testtools	Some (optional) utilities for use with testtools.
seaworthy.utils	

2.5.1 seaworthy

seaworthy

Todo: Write some API reference docs for *seaworthy*.

class DockerHelper(*namespace='test'*, *client=None*)

> **Todo:** Document this properly.

teardown()
Clean up all resources when we're done with them.

output_lines(*output*, *encoding='utf-8'*)
Convert bytestring container output or the result of a container exec command into a sequence of unicode lines.

Parameters

- **output** – Container output bytes or an `docker.models.containers.ExecResult` instance.
- **encoding** – The encoding to use when converting bytes to unicode (default `utf-8`).

Returns list[str]

wait_for_logs_matching(*container*, *matcher*, *timeout=10*, *encoding='utf-8'*, ***logs_kwargs*)
Wait for matching log line(s) from the given container by streaming the container's stdout and/or stderr outputs.

Each log line is decoded and any trailing whitespace is stripped before the line is matched.

Parameters

- **container** (*Container*) – Container who's log lines to wait for.
- **matcher** – Callable that returns True once it has matched a decoded log line(s).
- **timeout** – Timeout value in seconds.
- **encoding** – Encoding to use when decoding container output to strings.
- **logs_kwargs** – Additional keyword arguments to pass to `container.logs()`. For example, the `stdout` and `stderr` boolean arguments can be used to determine whether to stream stdout or stderr or both (the default).

Returns The final matching log line.

Raises

- **TimeoutError** – When the timeout value is reached before matching log lines have been found.
- **RuntimeError** – When all log lines have been consumed but matching log lines have not been found (the container must have stopped for its stream to have ended without error).

2.5.2 seaworthy.checks

Checks and test decorators for skipping tests that require Docker to be present.

docker_available()
Check if Docker is available and responsive.

docker_client()
A context manager that creates and cleans up a Docker API client.

In most cases, it's better to use *DockerHelper* instead.

dockertest()
Skip tests that require Docker to be available.

This is a function that returns a decorator so that we don't run arbitrary Docker client code on import. This implementation only works with tests based on `unittest.TestCase`. If you're using pytest, you probably want *`seaworthy.pytest.dockertest()`* instead.

2.5.3 seaworthy.client

A requests-based HTTP client for interacting with containers that have forwarded ports.

class ContainerHttpClient(*host*, *port*, *url_defaults=None*, *session=None*)
HTTP client for a specific container.

In most cases, these should be obtained from *`ContainerDefinition.http_client()`* instead of being instantiated directly.

close()
Closes the underlying Session object.

delete(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends a PUT request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.
- ****kwargs** – Optional arguments that `request` takes.

Returns response object

classmethod for_container(*container*, *container_port=None*)

Parameters

- **container** – The container to make requests against.
- **container_port** – The container port to make requests against. If `None`, the first container port is used.

Returns A ContainerClient object configured to make requests to the container.

get(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends a GET request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.
- ****kwargs** – Optional arguments that `request` takes.

Returns response object

head(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends a HEAD request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.

- ****kwargs** – Optional arguments that `request` takes.

Returns response object

options(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends an OPTIONS request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.
- ****kwargs** – Optional arguments that `request` takes.

Returns response object

patch(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends a PUT request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.
- ****kwargs** – Optional arguments that `request` takes.

Returns response object

post(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends a POST request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.
- ****kwargs** – Optional arguments that `request` takes.

Returns response object

put(*path=None*, *url_kwargs=None*, ***kwargs*)
Sends a PUT request.

Parameters

- **path** – The HTTP path (either absolute or relative).
- **url_kwargs** – Parameters to override in the generated URL. See *~hyperlink.URL*.
- ****kwargs** – Optional arguments that `request` takes.

Returns response object

request(*method*, *path=None*, *url_kwargs=None*, ***kwargs*)
Make a request against a container.

Parameters

- **method** – The HTTP method to use.
- **path** (*list*) – The HTTP path (either absolute or relative).
- **url_kwargs** (*dict*) – Parameters to override in the generated URL. See *~hyperlink.URL*.
- **kwargs** – Any other parameters to pass to Requests.

wait_for_response(*client*, *timeout*, *path='/'*, *expected_status_code=None*)
Try make a GET request with an HTTP client against a certain path and return once any response has been received, ignoring any errors.

Parameters

- **client** (ContainerHttpClient) – The HTTP client to use to connect to the container.
- **timeout** – Timeout value in seconds.
- **path** – HTTP path to request.
- **expected_status_code** (*int*) – If set, wait until a response with this status code is received. If not set, the status code will not be checked.

Raises **TimeoutError** – If a request fails to be made within the timeout period.

2.5.4 seaworthy.containers.nginx

class NginxContainer(*name='nginx'*, *image='nginx:alpine'*, ***kwargs*)
Nginx container definition.

base_kwargs()
Publish all exposed ports to the host.

exec_nginx(*args*)
Execute a `nginx` command inside a running container.

Params args a list of args for the command

exec_signal(*signal='reload'*)
Send a signal to the Nginx master process (`nginx -s`).

Parameters **signal** – one of: stop, quit, reopen, or reload

wait_for_start()
Wait for Nginx to return any valid HTTP response.

2.5.5 seaworthy.containers.postgresql

PostgreSQL container definition.

class PostgreSQLContainer(*name='postgresql'*, *image='postgres:alpine'*, *wait_patterns=('database system is ready to accept connections', 'database system is ready to accept connections')*, *database='database'*, *user='user'*, *password='password'*, ***kwargs*)
PostgreSQL container definition.

Todo: Write more docs.

---

base_kwargs()
Add a `tmpfs` entry for `/var/lib/postgresql/data` to avoid unnecessary disk I/O and `environment` entries for the configured db and user creds.

clean()
Remove all data by dropping and recreating the configured database.

Note: Only the configured database is removed. Any other databases remain untouched.

database_url()
Returns a "database URL" for use with DJ-Database-URL and similar libraries.

exec_pg_success(*cmd*)
Execute a command inside a running container as the postgres user, asserting success.

exec_psql(*command, psql_opts=['-qtA']*)
Execute a `psql` command inside a running container. By default the container's database is connected to.

Parameters

- **command** – the command to run (passed to `-c`)
- **psql_opts** – a list of extra options to pass to `psql`

Returns a tuple of the command exit code and output

list_databases()
Runs the `\list` command and returns a list of column values with information about all databases.

list_tables()
Runs the `\dt` command and returns a list of column values with information about all tables in the database.

list_users()
Runs the `\du` command and returns a list of column values with information about all user roles.

2.5.6 seaworthy.containers.rabbitmq

RabbitMQ container definition.

class RabbitMQContainer(*name='rabbitmq', image='rabbitmq:alpine', wait_patterns=('Server startup complete',), vhost='/vhost', user='user', password='password', **kwargs*)
RabbitMQ container definition.

Todo: Write more docs.

base_kwargs()
Add a `tmpfs` entry for `/var/lib/rabbitmq` to avoid unnecessary disk I/O and `environment` entries for the configured vhost and user creds.

broker_url()
Returns a "broker URL" for use with Celery.

clean()
Remove all data by using `rabbitmqctl` to eval `rabbit_mnesia:reset()`.

exec_rabbitmqctl(*command, args=[], rabbitmqctl_opts=['-q']*)
Execute a `rabbitmqctl` command inside a running container.

Parameters

- **command** – the command to run
- **args** – a list of args for the command

- **rabbitmqctl_opts** – a list of extra options to pass to `rabbitmqctl`

Returns a tuple of the command exit code and output

exec_rabbitmqctl_list(*resources, args=[], rabbitmq_opts=['-q', '–no-table-headers']*)
Execute a `rabbitmqctl` command to list the given resources.

Parameters

- **resources** – the resources to list, e.g. `'vhosts'`
- **args** – a list of args for the command
- **rabbitmqctl_opts** – a list of extra options to pass to `rabbitmqctl`

Returns a tuple of the command exit code and output

list_queues()
Run the `list_queues` command (for the default vhost) and return a list of tuples describing the queues.

Returns A list of 2-element tuples. The first element is the queue name, the second is the current queue size.

list_users()
Run the `list_users` command and return a list of tuples describing the users.

Returns A list of 2-element tuples. The first element is the username, the second a list of tags for the user.

list_vhosts()
Run the `list_vhosts` command and return a list of vhost names.

2.5.7 seaworthy.containers.redis

Redis container definition.

class RedisContainer(*name='redis', image='redis:alpine', wait_patterns=('* Ready to accept connections',), **kwargs*)
Redis container definition.

Todo: Write more docs.

---

base_kwargs()
Add a `tmpfs` entry for `/data` to avoid unnecessary disk I/O.

clean()
Remove all data by sending the `FLUSHALL` command.

exec_redis_cli(*command, args=[], db=0, redis_cli_opts=[]*)
Execute a `redis-cli` command inside a running container.

Parameters

- **command** – the command to run
- **args** – a list of args for the command
- **db** – the db number to query (default `0`)
- **redis_cli_opts** – a list of extra options to pass to `redis-cli`

Returns a tuple of the command exit code and output

list_keys(*pattern='*', db=0*)
Run the `KEYS` command and return the list of matching keys.

Parameters

- **pattern** – the pattern to filter keys by (default `*`)
- **db** – the db number to query (default `0`)

2.5.8 seaworthy.definitions

Wrappers over Docker resource types to aid in setup/teardown of and interaction with Docker resources.

class ContainerDefinition(*name*, *image*, *wait_patterns=None*, *wait_timeout=None*, *create_kwargs=None*, *helper=None*)
This is the base class for container definitions. Instances (and instances of subclasses) are intended to be used both as test fixtures and as convenient objects for operating on containers being tested.

Todo: Document this properly.

A container object may be used as a context manager to ensure proper setup and teardown of the container around the code that uses it:

```
with ContainerDefinition('my_container', IMAGE, helper=ch) as c:
    assert c.status() == 'running'
```

(Note that this only works if the container has a helper set and does not have a container created.)

as_fixture(*name=None*)
A decorator to inject this container into a function as a test fixture.

base_kwargs()
Override this method to provide dynamically generated base kwargs for the resource.

clean()
This method should "clean" the container so that it is in the same state as it was when it was started. It is up to the implementer of this method to decide how the container should be cleaned. See *clean_container_fixtures()* for how this can be used with pytest fixtures.

create(***kwargs*)
Create an instance of this resource definition.

Only one instance may exist at any given time.

get_first_host_port()
Get the first mapping of the first (lowest) container port that has a mapping. Useful when a container publishes only one port.

Note that unlike the Docker API, which sorts ports lexicographically (e.g. `90/tcp` > `8000/tcp`), we sort ports numerically so that the lowest port is always chosen.

get_host_port(*container_port*, *proto='tcp'*, *index=0*)

Parameters

- **container_port** – The container port.
- **proto** – The protocol ('tcp' or 'udp').
- **index** – The index of the mapping entry to return.

Returns A tuple of the interface IP and port on the host.

get_logs(*stdout=True*, *stderr=True*, *timestamps=False*, *tail='all'*, *since=None*)
Get container logs.

This method does not support streaming, use *stream_logs()* for that.

halt(*stop_timeout=5*)
Stop the container and remove it. The opposite of *run()*.

http_client(*port=None*)
Construct an HTTP client for this container.

inner()

Returns the underlying Docker model object

merge_kwargs(*default_kwargs*, *kwargs*)
Override this method to merge kwargs differently.

ports
The ports (exposed and published) of the container.

pytest_clean_fixtures(*name*, *scope='function'*, *dependencies=()*)
Creates a pytest fixture for a container that can be "cleaned". See *clean_container_fixtures()*.

Note: This method returns two fixture functions. It is important to keep references to the returned functions within the scope of the tests that use the fixtures.

Note: This method is only available if pytest is used.

Parameters

- **name** – The fixture name.
- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

pytest_fixture(*name*, *scope='function'*, *dependencies=()*)
Create a pytest fixture for the resource. See *resource_fixture()*.

Note: This method returns a fixture function. It is important to keep a reference to the returned function within the scope of the tests that use the fixture.

Note: This method is only available if pytest is used.

Parameters

- **name** – The fixture name.
- **scope** – The scope of the fixture.

- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

remove(***kwargs*)
Remove an instance of this resource definition.

run(*fetch_image=True*, ***kwargs*)
Create the container and start it. Similar to `docker run`.

Parameters

- **fetch_image** – Whether to try pull the image if it's not found. The behaviour here is similar to `docker run` and this parameter defaults to `True`.
- ****kwargs** – Keyword arguments passed to *create()*.

set_helper(*helper*)

Todo: Document this.

---

setup(*helper=None*, ***run_kwargs*)
Creates the container, starts it, and waits for it to completely start.

Parameters

- **helper** – The resource helper to use, if one was not provided when this container definition was created.
- ****run_kwargs** – Keyword arguments passed to *run()*.

Returns

This container definition instance. Useful for creating and setting up a container in a single step:

```
con = ContainerDefinition('conny', 'nginx').setup(helper=dh)
```

start()
Start the container. The container must have been created.

status()
Get the container's current status from Docker.

If the container does not exist (before creation and after removal), the status is `None`.

stop(*timeout=5*)
Stop the container. The container must have been created.

Parameters **timeout** – Timeout in seconds to wait for the container to stop before sending a `SIGKILL`. Default: 5 (half the Docker default)

stream_logs(*stdout=True*, *stderr=True*, *tail='all'*, *timeout=10.0*)
Stream container output.

teardown()
Stop and remove the container if it exists.

wait_for_logs_matching(*matcher*, *timeout=10*, *encoding='utf-8'*, ***logs_kwargs*)
Wait for logs matching the given matcher.

wait_for_start()
Wait for the container to start.

By default this will wait for the log lines matching the patterns passed in the `wait_patterns` parameter of the constructor using an UnorderedMatcher. For more advanced checks for container startup, this method should be overridden.

class NetworkDefinition(*name*, *create_kwargs=None*, *helper=None*)
This is the base class for network definitions.

Todo: Document this properly.

---

as_fixture(*name=None*)
A decorator to inject this container into a function as a test fixture.

base_kwargs()
Override this method to provide dynamically generated base kwargs for the resource.

create(***kwargs*)
Create an instance of this resource definition.

Only one instance may exist at any given time.

inner()

Returns the underlying Docker model object

merge_kwargs(*default_kwargs*, *kwargs*)
Override this method to merge kwargs differently.

pytest_fixture(*name*, *scope='function'*, *dependencies=()*)
Create a pytest fixture for the resource. See *resource_fixture()*.

Note: This method returns a fixture function. It is important to keep a reference to the returned function within the scope of the tests that use the fixture.

Note: This method is only available if pytest is used.

Parameters

- **name** – The fixture name.
- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

remove(***kwargs*)
Remove an instance of this resource definition.

set_helper(*helper*)

Todo: Document this.

---

setup(*helper=None*, ***create_kwargs*)
Setup this resource so that is ready to be used in a test. If the resource has already been created, this call does nothing.

For most resources, this just involves creating the resource in Docker.

Parameters

- **helper** – The resource helper to use, if one was not provided when this resource definition was created.
- ****create_kwargs** – Keyword arguments passed to *create()*.

Returns

This definition instance. Useful for creating and setting up a resource in a single step:

```
volume = VolumeDefinition('volly').setup(helper=docker_helper)
```

teardown()
Teardown this resource so that it no longer exists in Docker. If the resource has already been removed, this call does nothing.

For most resources, this just involves removing the resource in Docker.

class VolumeDefinition(*name*, *create_kwargs=None*, *helper=None*)
This is the base class for volume definitions.

The following is an example of how `VolumeDefinition` can be used to attach volumes to a container:

```
from seaworthy.definitions import ContainerDefinition

class DjangoContainer(ContainerDefinition):
    IMAGE = "seaworthy-demo:django"
    WAIT_PATTERNS = (r"Booting worker",)

    def __init__(self, name, socket_volume, static_volume, db_url):
        super().__init__(name, self.IMAGE, self.WAIT_PATTERNS)
        self.socket_volume = socket_volume
        self.static_volume = static_volume
        self.db_url = db_url

    def base_kwargs(self):
        return {
            "volumes": {
                self.socket_volume.inner(): "/var/run/gunicorn",
                self.static_volume.inner(): "/app/static:ro",
            },
            "environment": {"DATABASE_URL": self.db_url}
        }

# Create definition instances
socket_volume = VolumeDefinition("socket")
static_volume = VolumeDefinition("static")
django_container = DjangoContainer(
    "django", socket_volume, static_volume,
    postgresql_container.database_url())

# Create pytest fixtures
socket_volume_fixture = socket_volume.pytest_fixture("socket_volume")
static_volume_fixture = static_volume.pytest_fixture("static_volume")
```

(continues on next page)

(continued from previous page)

```
django_fixture = django_container.pytest_fixture(
    "django_container",
    dependencies=[
        "socket_volume", "static_volume", "postgresql_container"])
```

This example is explained in the introductory blog post and demo repository.

Todo: Document this properly.

---

as_fixture(*name=None*)
A decorator to inject this container into a function as a test fixture.

base_kwargs()
Override this method to provide dynamically generated base kwargs for the resource.

create(***kwargs*)
Create an instance of this resource definition.

Only one instance may exist at any given time.

inner()

Returns the underlying Docker model object

merge_kwargs(*default_kwargs*, *kwargs*)
Override this method to merge kwargs differently.

pytest_fixture(*name*, *scope='function'*, *dependencies=()*)
Create a pytest fixture for the resource. See *resource_fixture()*.

Note: This method returns a fixture function. It is important to keep a reference to the returned function within the scope of the tests that use the fixture.

Note: This method is only available if pytest is used.

Parameters

- **name** – The fixture name.
- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

remove(***kwargs*)
Remove an instance of this resource definition.

set_helper(*helper*)

Todo: Document this.

---

setup(*helper=None*, ***create_kwargs*)
Setup this resource so that is ready to be used in a test. If the resource has already been created, this call does nothing.

For most resources, this just involves creating the resource in Docker.

Parameters

- **helper** – The resource helper to use, if one was not provided when this resource definition was created.
- ****create_kwargs** – Keyword arguments passed to *create()*.

Returns

This definition instance. Useful for creating and setting up a resource in a single step:

```
volume = VolumeDefinition('volly').setup(helper=docker_helper)
```

teardown()
Teardown this resource so that it no longer exists in Docker. If the resource has already been removed, this call does nothing.

For most resources, this just involves removing the resource in Docker.

deep_merge(**dicts*)
Recursively merge all input dicts into a single dict.

2.5.9 seaworthy.helpers

Classes that track resource creation and removal to ensure that all resources are namespaced and cleaned up after use.

class ContainerHelper(*client*, *namespace*, *image_helper*, *network_helper*, *volume_helper*)

Todo: Document this properly.

---

create(*name*, *image*, *fetch_image=False*, *network=None*, *volumes={}*, ***kwargs*)
Create a new container.

Parameters

- **name** – The name for the container. This will be prefixed with the namespace.
- **image** – The image tag or image object to create the container from.
- **network** – The network to connect the container to. The container will be given an alias with the `name` parameter. Note that, unlike the Docker Python client, this parameter can be a `Network` model object, and not just a network ID or name.
- **volumes** – A mapping of volumes to bind parameters. The keys to this mapping can be any of three types of objects:
  - A `Volume` model object
  - The name of a volume (str)
  - A path on the host to bind mount into the container (str)

  The bind parameters, i.e. the values in the mapping, can be of two types:
  - A full bind specifier (dict), for example `{'bind': '/mnt', 'mode': 'rw'}`

  - A "short-form" bind specifier (str), for example `/mnt:rw`
- **fetch_image** – Whether to attempt to pull the image if it is not found locally.
- **kwargs** – Other parameters to create the container with.

remove(*container*, *force=True*, *volumes=True*)
Remove a container.

Parameters

- **container** – The container to remove.
- **force** – Whether to force the removal of the container, even if it is running. Note that this defaults to True, unlike the Docker default.
- **volumes** – Whether to remove any volumes that were created implicitly with this container, i.e. any volumes that were created due to `VOLUME` directives in the Dockerfile. External volumes that were manually created will not be removed. Note that this defaults to True, unlike the Docker default (where the equivalent parameter, `v`, defaults to False).

class DockerHelper(*namespace='test'*, *client=None*)

Todo: Document this properly.

---

teardown()
Clean up all resources when we're done with them.

class ImageHelper(*client*)

Todo: Document this properly.

---

fetch(*tag*)
Fetch this image if it isn't already present.

class NetworkHelper(*client*, *namespace*)

Todo: Document this properly.

---

create(*name*, *check_duplicate=True*, ***kwargs*)
Create a new network.

Parameters

- **name** – The name for the network. This will be prefixed with the namespace.
- **check_duplicate** – Whether or not to check for networks with the same name. Docker allows the creation of multiple networks with the same name (unlike containers). This seems to cause problems sometimes for some reason (?). The Docker Python client _claims_ (as of 2.5.1) that `check_duplicate` defaults to True but it actually doesn't. We default it to True ourselves here.
- **kwargs** – Other parameters to create the network with.

get_default(*create=True*)
Get the default bridge network that containers are connected to if no other network options are specified.

Parameters **create** – Whether or not to create the network if it doesn't already exist.

remove(*resource*, ***kwargs*)
Remove an instance of this resource type.

class VolumeHelper(*client*, *namespace*)

Todo: Document this properly.

---

create(*name*, ***kwargs*)
Create a new volume.

Parameters

- **name** – The name for the volume. This will be prefixed with the namespace.
- **kwargs** – Other parameters to create the volume with.

remove(*resource*, ***kwargs*)
Remove an instance of this resource type.

fetch_image(*client*, *name*)
Fetch an image if it isn't already present.

This works like `docker pull` and will pull the tag `latest` if no tag is specified in the image name.

fetch_images(*client*, *images*)
Fetch images if they aren't already present.

2.5.10 seaworthy.ps

Tools for asserting on processes running in containers using `ps`.

build_process_tree(*ps_rows*)
Build a tree structure from a list of PsRow objects. :param ps_rows: a list of PsRow objects :return: a PsTree object

list_container_processes(*container*)
List the processes running inside a container. We use an exec rather than *container.top()* because we want to run 'ps' inside the container. This is because we want to get PIDs and usernames in the container's namespaces. *container.top()* uses 'ps' from outside the container in the host's namespaces. Note that this requires the container to have a 'ps' that responds to the arguments we give it– we use BusyBox's (Alpine's) 'ps' as a baseline for available functionality. :param container: the container to query :return: a list of PsRow objects

exception PsException
Exception indicating problems operating on process lists and trees.

class PsRow(*pid*, *ppid*, *ruser*, *args*)
Representation of a process list entry, containing the details of a single process.

classmethod columns()
List the columns required to construct a suitable `ps` command.

class PsTree(*row*, *children=NOTHING*)
Node in a process tree, linking a *PsRow* to its child processes.

count()
Return the number of processes in this subtree.

2.5.11 seaworthy.pytest

Some (optional) utilities for use with pytest.

While Seaworthy doesn't require pytest, we find it useful in downstream container tests we write with Seaworthy. This module contains various bits and pieces to make Seaworthy work better with pytest.

dockertest()
Skip tests that require Docker to be available.

This is a function that returns a decorator so that we don't run arbitrary Docker client code on import. Unlike *seaworthy.checks.dockertest()*, this implementation doesn't require `unittest.TestCase`. It does, however, require pytest.

2.5.12 seaworthy.pytest.checks

pytest mark to skip tests that require Docker.

dockertest()
Skip tests that require Docker to be available.

This is a function that returns a decorator so that we don't run arbitrary Docker client code on import. Unlike *seaworthy.checks.dockertest()*, this implementation doesn't require `unittest.TestCase`. It does, however, require pytest.

2.5.13 seaworthy.pytest.fixtures

A number of pytest fixtures or factories for fixtures.

clean_container_fixtures(*container*, *name*, *scope='class'*, *dependencies=()*)
Creates a fixture for a container that can be "cleaned". When a code block is marked with `@pytest.mark.clean_<fixture name>` then the `clean` method will be called on the container object before it is passed as an argument to the test function.

Note: This function returns two fixture functions. It is important to keep references to the returned functions within the scope of the tests that use the fixtures.

```
f1, f2 = clean_container_fixtures(PostgreSQLContainer(), 'postgresql')

class TestPostgresqlContainer
    @pytest.mark.clean_postgresql
    def test_clean_container(self, web_container, postgresql):
        """
        Test something about the container that requires it to have a
        clean state (e.g. database table creation).
        """

    def test_dirty_container(self, web_container, postgresql):
        """
        Test something about the container that doesn't require it to
```

(continues on next page)

(continued from previous page)

```
        have a clean state (e.g. testing something about a dependent
        container).
        """
```

Parameters

- **container** – A "container" object that is a subclass of *ContainerDefinition*.
- **name** – The fixture name.
- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

Returns A tuple of two fixture functions.

docker_helper()
Default fixture for *DockerHelper*. Has module scope.

docker_helper_fixture(*name='docker_helper'*, *scope='module'*, ***kwargs*)
Create a fixture for *DockerHelper*.

This can be used to create a fixture with a different name to the default. It can also be used to override the scope of the default fixture:

```
docker_helper = docker_helper_fixture(scope='class')
```

Parameters

- **name** – The name of the fixture.
- **scope** – The scope of the fixture.
- **kwargs** – Keyword arguments to pass to the *DockerHelper* constructor.

image_fetch_fixture(*image*, *name*, *scope='module'*)
Create a fixture to fetch an image.

resource_fixture(*definition*, *name*, *scope='function'*, *dependencies=()*)
Create a fixture for a resource.

Note: This function returns a fixture function. It is important to keep a reference to the returned function within the scope of the tests that use the fixture.

```
fixture = resource_fixture(PostgreSQLContainer(), 'postgresql')

def test_container(postgresql):
    """Test something about the PostgreSQL container..."""
```

Parameters

- **definition** – A resource definition, one of those defined in the *seaworthy.definitions* module.
- **name** – The fixture name.

- **scope** – The scope of the fixture.
- **dependencies** – A sequence of names of other pytest fixtures that this fixture depends on. These fixtures will be requested from pytest and so will be setup, but nothing is done with the actual fixture values.

Returns The fixture function.

2.5.14 seaworthy.stream

stream_timeout(*stream*, *timeout*, *timeout_msg=None*)
Iterate over items in a streaming response from the Docker client within a timeout.

Parameters

- **stream** (*CancellableStream*) – Stream from the Docker client to consume items from.
- **timeout** – Timeout value in seconds.
- **timeout_msg** – Message to raise in the exception when a timeout occurs.

2.5.15 seaworthy.stream.logs

stream_logs(*container*, *timeout=10.0*, ***logs_kwargs*)
Stream logs from a Docker container within a timeout.

Parameters

- **container** (*Container*) – Container who's log lines to stream.
- **timeout** – Timeout value in seconds.
- **logs_kwargs** – Additional keyword arguments to pass to `container.logs()`. For example, the `stdout` and `stderr` boolean arguments can be used to determine whether to stream stdout or stderr or both (the default).

Raises **TimeoutError** – When the timeout value is reached before the logs have completed.

wait_for_logs_matching(*container*, *matcher*, *timeout=10*, *encoding='utf-8'*, ***logs_kwargs*)
Wait for matching log line(s) from the given container by streaming the container's stdout and/or stderr outputs.

Each log line is decoded and any trailing whitespace is stripped before the line is matched.

Parameters

- **container** (*Container*) – Container who's log lines to wait for.
- **matcher** – Callable that returns True once it has matched a decoded log line(s).
- **timeout** – Timeout value in seconds.
- **encoding** – Encoding to use when decoding container output to strings.
- **logs_kwargs** – Additional keyword arguments to pass to `container.logs()`. For example, the `stdout` and `stderr` boolean arguments can be used to determine whether to stream stdout or stderr or both (the default).

Returns The final matching log line.

Raises

- **TimeoutError** – When the timeout value is reached before matching log lines have been found.
- **RuntimeError** – When all log lines have been consumed but matching log lines have not been found (the container must have stopped for its stream to have ended without error).

2.5.16 seaworthy.stream.matchers

class CombinationMatcher(**matchers*)
Matcher that combines multiple input matchers.

classmethod by_equality(**expected_items*)
Construct an instance of this combination matcher from a list of expected items and/or StreamMatcher instances.

classmethod by_regex(**patterns*)
Construct an instance of this combination matcher from a list of regex patterns and/or StreamMatcher instances.

class EqualsMatcher(*expected_item*)
Matcher that matches items by equality.

args_str()
Return an args string for the repr.

match(*item*)
Return `True` if the item matches the expected value exactly, otherwise `False`.

class OrderedMatcher(**matchers*)
Matcher that takes a list of matchers, and uses one after the next after each has a successful match. Returns True ("matches") on the final match.

Note: This is a *stateful* matcher. Once it has done its matching, you'll need to create a new instance.

args_str()
Return an args string for the repr.

match(*item*)
Return `True` if the expected matchers are matched in the expected order, otherwise `False`.

class RegexMatcher(*pattern*)
Matcher that matches items by regex pattern.

args_str()
Return an args string for the repr.

match(*item*)
Return `True` if the item matches the expected regex, otherwise `False`.

class StreamMatcher
Abstract base class for stream matchers.

args_str()
Return an args string for the repr.

match(*item*)
Return `True` if the matcher matches an item, otherwise `False`.

class UnorderedMatcher(**matchers*)
Matcher that takes a list of matchers, and matches each one to an item. Each item is tested against each unmatched matcher until a match is found or all unmatched matchers are checked. Returns True ("matches") on the final match.

Note: This is a *stateful* matcher. Once it has done its matching, you'll need to create a new instance.

args_str()
Return an args string for the repr.

match(*item*)
Return `True` if the expected matchers are matched in any order, otherwise `False`.

to_matcher(*matcher_factory*, *obj*)
Turn an object into a *StreamMatcher* unless it already is one.

Parameters

- **matcher_factory** – A callable capable of turning *obj* into a *StreamMatcher*.
- **obj** – A *StreamMatcher* or an object to turn into one.

Returns *StreamMatcher*

2.5.17 seaworthy.testtools

Some (optional) utilities for use with testtools.

While Seaworthy doesn't require testtools, we find it useful in downstream container tests we write with Seaworthy. This module contains various bits and pieces to make Seaworthy work better with testtools.

class MatchesPsTree(*ruser*, *args*, *pid=None*, *ppid=None*, *children=()*)
Matches a nested PsTree object in a sensible way.

The *ruser* and *args* fields are always checked. The *pid* and *ppid* fields are only checked if non-None values are explicitly provided, because real pids are essentially arbitrary integers. The *children* field is always checked, but order is ignored.

match(*value*)
Return `None` if this matcher matches something, a *PsTreeMismatch* otherwise.

class PsTreeMismatch(*row_fields*, *child_count*, *fields_mm*, *children_mm*)
Custom mismatch container for MatchesPsTree so we get somewhat more comprehensible failure messages.

describe()
Describe the mismatch.

2.5.18 seaworthy.utils

output_lines(*output*, *encoding='utf-8'*)
Convert bytestring container output or the result of a container exec command into a sequence of unicode lines.

Parameters

- **output** – Container output bytes or an `docker.models.containers.ExecResult` instance.
- **encoding** – The encoding to use when converting bytes to unicode (default `utf-8`).

Returns list[str]

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

T

U

V

W