CA650 Software Process Quality **Anthony Troy**
XUnit Report Date: 25/2/2016

Report Requirements

Using any XUnit-style framework, implement a class for creating and adding integers to a Stack with a corresponding test suite. Typically you should have create, push, pop and isEmpty methods.

Attachments

Convenient browsing of the discussed source code in this report is available through the following GitHub repository: full-of-foo/pystack. The involved files of note are as follows:

- *pystack/__init__.py*
- *pystack/exc.py* (package exception declarations)
- *pystack/stack.py* (stack implementation module)
- *tests/__init__.py*
- *tests/fixtures.py* (test fixture module)
- *tests/test_stack.py* (test suite for the stack)
- *report/report.pdf*
- *report/report.tex* (LaTeX file for this report)
- *setup.py* (python package setup module)
- *test_requirements.txt* (listing for test requirement packages)
- *tox.ini* (test runner configuration file)
- *circle.yml* (continuous integration service configuration file)

Stack Implementation

Under Python 3.5, we implement our list-based stack as follows:

```
from pystack.exc import EmptyStackError


class Stack(object):
    """A list-based stack implementation.

    Attributes:
        _data (list): Private list maintaining stack elements
    """

    def __init__(self):
        """Initialises an empty stack."""
        self._data = []
```

```
    def is_empty(self):
        """Checks if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self._data) == 0

    def push(self, item):
        """Pushes an item onto the stack.

        Returns:
            The pushed item.
        """
        self._data.append(item)
        return item

    def pop(self):
        """Removes and returns the item ontop of the stack.

        Returns:
            The last item added to the stack.
        Raises:
            EmptyStackError: given the an already empty stack.
        """
        if self.is_empty():
            raise EmptyStackError()
        item = self._data[len(self._data) - 1]
        del self._data[len(self._data) - 1]
        return item


class IntStack(Stack):
    """A list-based stack implementation for `int`s."""

    def push(self, item: int) -> int:
        """Pushes an `int` item onto the stack.

        Returns:
            The pushed `int`.
        Raises:
            TypeError: given the item is not an `int`.
        """
        if not isinstance(item, int):
            raise TypeError()
        return super().push(item)

    def pop(self) -> int:
        return super().pop()
```

Test Suite Implementation

Using unittest, Python's native XUnit framework, we implement our stack test suite as follows:

```
import unittest
from pystack.stack import IntStack
from pystack.exc import EmptyStackError
from tests.fixtures import BAD_INT_STACK_VALUES, build_int_stack
```

```
class TestIntStack(unittest.TestCase):
    """Test cases for the `IntStack` implementation.

    Fixtures:
        stack (IntStack): a populated `IntStack` instance
        emptyStack (IntStack): an empty `IntStack` instance
    """

    def setUp(self):
        self.stack = build_int_stack()
        self.emptyStack = build_int_stack([])

    def test_init(self):
        self.assertEqual(len(IntStack()._data), 0)

    def test_is_empty(self):
        self.assertTrue(self.emptyStack.is_empty())
        self.assertFalse(self.stack.is_empty())

    def test_cannot_push_other_types(self):
        [self.assertRaises(TypeError, self.stack.push, v) for v in
    BAD_INT_STACK_VALUES]

    def test_cannot_pop_when_empty(self):
        self.assertRaises(EmptyStackError, self.emptyStack.pop)

    def test_postioning_with_push(self):
        self.assertEqual(self.stack.push(3), 3)
        self.assertEqual(self.stack._data[-1:][0], 3)

        self.assertEqual(self.stack.push(4), 4)
        self.assertEqual(self.stack._data[-1:][0], 4)
        self.assertEqual(self.stack._data[-2:][0], 3)

    def test_postioning_with_pop(self):
        self.assertEqual(self.stack.pop(), 2)
        self.assertEqual(self.stack._data[-1:][0], 1)

        self.assertEqual(self.stack.pop(), 1)
        self.assertEqual(len(self.stack._data), 0)
```

Test Runner Implementation

Leveraging tox, an open-source packaging and test runner framework for Python, we implement the following test run tasks:

```
[tox]
envlist = py35

[testenv]
basepython = python3.5
setenv =
  PYTHONHASHSEED = 100
deps=
 -rtest_requirements.txt
commands =
  {envbindir}/python -m unittest {posargs}
```

```
[testenv:coverage]
commands =
  {envbindir}/coverage erase
  {envbindir}/coverage run -m unittest {posargs}
  {envbindir}/coverage combine
  {envbindir}/coverage report -m --omit="tests/*"
```

Test Suite Execution

As entailed above, our default task simply runs all unittest test suites in the current package. The output of such a run is as follows:

```
~/dev/pystack > $ tox
GLOB sdist-make: ~/dev/pystack/setup.py
py35 create: ~/dev/pystack/.tox/py35
py35 installdeps: -rtest_requirements.txt
py35 inst: ~/dev/pystack/.tox/dist/pystack-1.0.0.zip
py35 installed: appnope==0.1.0,coverage==4.0.3,decorator==4.0.9,
      gnureadline==6.3.3,ipdb==0.8.2,ipython==4.1.1,ipython-genutils==0.1.0,
      path.py==8.1.2,pep8==1.7.0,pexpect==4.0.1,pickleshare==0.6,
      ptyprocess==0.5.1,pystack==1.0.0,simplegeneric==0.8.1,
      traitlets==4.1.0,wheel==0.29.0
py35 runtests: PYTHONHASHSEED='100'
py35 runtests: commands[0] | ~/dev/pystack/.tox/py35/bin/python -m unittest
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
------------------------------ summary -------------------------------
  py35: commands succeeded
  congratulations :)
```

Local Coverage Execution

Our second task, named "coverage", executes a popular Python line coverage tool against our source code. The output of this coverage run is as follows:

```
~/dev/pystack > $ tox -e coverage
GLOB sdist-make: ~/dev/pystack/setup.py
coverage create: ~/dev/pystack/.tox/coverage
...
coverage runtests: commands[0] | ~/dev/pystack/.tox/coverage/bin/coverage erase
coverage runtests: commands[1] | ~/dev/pystack/.tox/coverage/bin/coverage run -m
    unittest
coverage runtests: commands[2] | ~/dev/pystack/.tox/coverage/bin/coverage combine
coverage runtests: commands[3] | ~/dev/pystack/.tox/coverage/bin/coverage report
    -m --omit=tests/*
Name                    Stmts   Miss  Cover   Missing
-----------------------------------------------------
pystack/__init__.py         0      0   100%
pystack/exc.py              1      0   100%
pystack/stack.py           22      0   100%
-----------------------------------------------------
TOTAL                      23      0   100%
```

Automated Testing Implementation

Our pystack project avails of a continuous integration service named CircleCI. The service will build and execute our test suite upon the pushing of a candidate build to a given remote GitHub branch. One can browse these builds at circleci.com/gh/full-of-foo/pystack. Our configuration for the service is as follows:

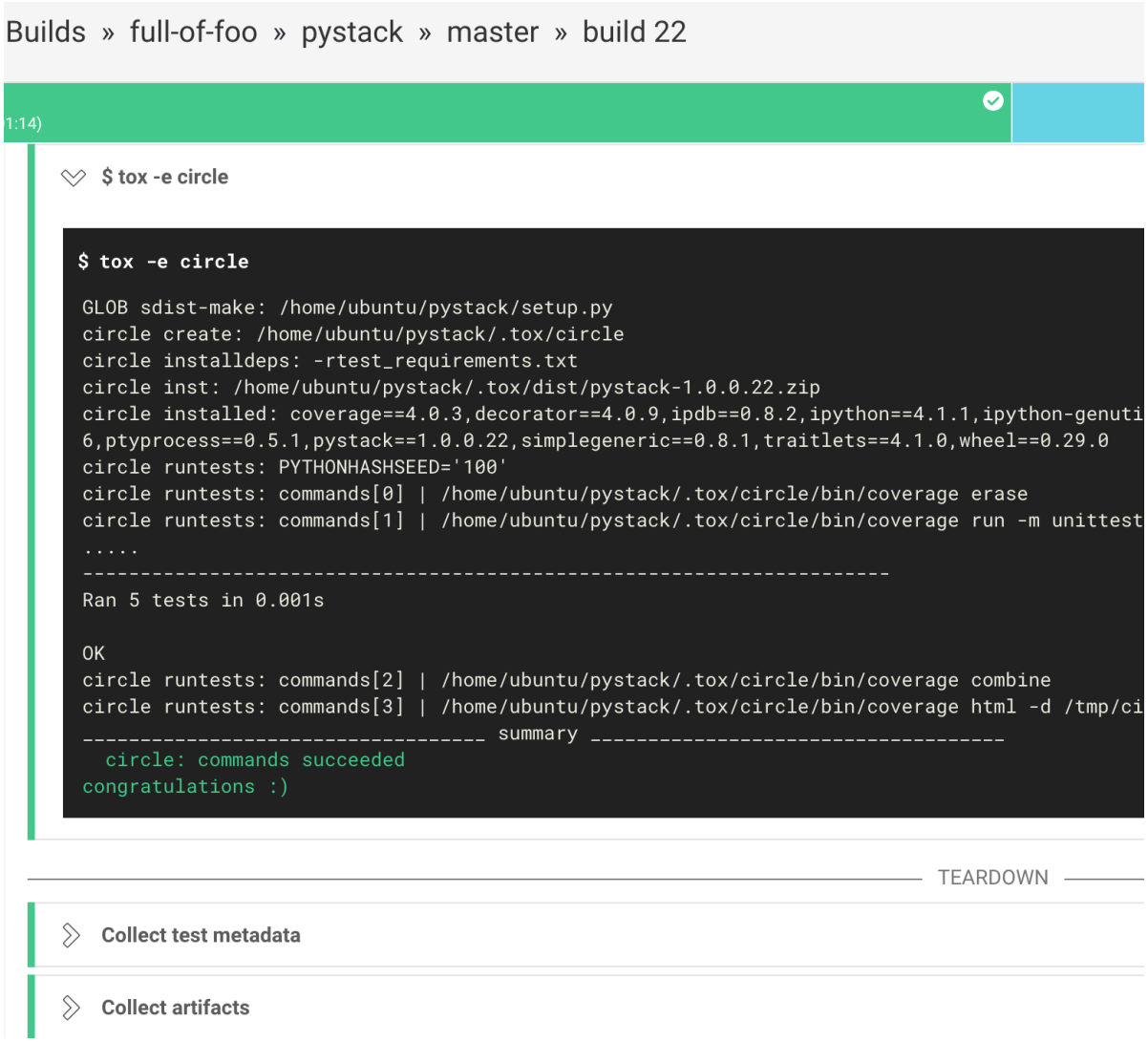
```
machine:
  python:
    version: 3.5.0

dependencies:
  override:
    - pip install tox

test:
  override:
    - tox -e circle
```

In a production scenario there might be an extra step in this configuration that deploys our pystack package to a server given that the build, and in turn tests, have passed. We have defined a new run task that will execute our test suite and also generate a HTML report of our line coverage which is persisted by the CircleCI service per build. By doing so during the build process we can ensure that coverage in the project will be maintained, as builds not reaching 100% coverage will fail. This "circle" run task is as follows:

```
[testenv:circle]
passenv =
  CIRCLE_ARTIFACTS
commands =
  {envbindir}/coverage erase
  {envbindir}/coverage run -m unittest {posargs}
  {envbindir}/coverage combine
  {envbindir}/coverage html -d {env:CIRCLE_ARTIFACTS:}/coverage --omit="tests/*"
```

Automated Test Execution

Upon pushing a commit to our remote GitHub master branch we trigger a new build on CircleCI. Some example output from this successful build is as follows:



When this successful build completes CircleCI makes our build artefacts available. These include a coverage/index.html file which allows us browse the coverage report for the given build: