PyHamcrest: Check What You Want to Check

Moshe Zadka – https://cobordism.com

2020

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)
Ancestral homeland of the Ramaytush Ohlone

What Is a Unit Test?

What Is a Unit Test?

Isolation?

What Is a Unit Test?

Isolation?
Fast?

What Is a Unit Test?

Isolation?
Fast?
Small System-Under-Test?

Python Unit Test Anatomy

- Runner (nose, pytest, virtue, etc.)
- Test case (unittest.TestCase, testtools.TestCase, etc.)
- Assertions (pytest assert rewriting, `unittest.TestCase.assert_*`, `|testtools.TestCase.assert_that|`)

All-in-one or A-la-carte?

- All in one: pytest, testtools, unittest
- A-la-carte runners: nose, virtue
- A-la-carte assertions: hamcrest

Hamcrest Composability

- Works in any runner
- Works with any test case
- Focuses on just assertions

Bad Unit Test One: False Alarm

AKA: False positive, Type I Error, Boy Who Cried Wolf
Asserting things that don't have to be true

Bad Unit Test Two: Missing Alarm

AKA: False negative, Type II Error, Boy Who Cried Wolf But Nobody Believed Him
Not asserting things that have to be true

Unit Test (Suite) Value

```
def f_score(
        beta,          # 1 if False Alarms /
                       # Missing Alarms
                       # are equally bad
                       # Other common values:
                       # 2 (Missing Alarms
                       # matter more),
                       # 0.5 (False Alarms
                       # matter more)
        true_alarm,    # Test runs that
                       # caught a bug
        false_alarm,   # Test runs that
                       # failed without a bug
        missing_alarm, # Bugs not caught
    ):
    ...
```

Unit Test (Suite) Value

```python
def f_score(
        beta,
        true_alarm,
        false_alarm,
        missing_alarm,
    ):
    numerator = (1 + beta**2) * true_alarm
    denominator = (
        numerator +
        beta**2 * missing_alarm + false_alarm
    )
    return numerator / denominator
```

Equality

```
with show_assert():
    assert_that(1, equal_to(2))
```

```
Expected: <2>
     but: was <1>
```

Containment

```
with show_assert():
    assert_that([1, 2, 3], has_item(5))

Expected: a sequence containing <5>
     but: was <[1, 2, 3]>
```

Any

```
with show_assert():
    assert_that(
        1,
        any_of(
          equal_to(2),
          equal_to(0),
        )
    )
```

```
Expected: (<2> or <0>)
     but: was <1>
```

All

```
with show_assert():
    assert_that(
        [1, 2, 3],
        all_of(
          has_item(1),
          has_item(4),
        )
    )
```

```
Expected: (a sequence containing <1> and a sequence containing <4>)
     but: a sequence containing <4> was <[1, 2, 3]>
```

Compose

```
with show_assert():
    assert_that(
        [[1, 2], [3, 4]],
        has_item(
          has_item(5),
        )
    )

Expected: a sequence containing a sequence
containing <5>
     but: was <[[1, 2], [3, 4]]>
```

Order

```
with show_assert():
    assert_that(
        [1, 2, 3],
        contains_exactly(1, 2, 4)
    )
```

```
Expected: a sequence containing [<1>, <2>, <4>]
     but: item 2: was <3>
```

Any Order

```
with show_assert():
    assert_that(
        [1, 2, 3],
        contains_inanyorder(4, 3, 1)
    )

Expected: a sequence over [<4>, <3>, <1>] in any
order
     but: not matched: <2>
```

Boolean Expressions

```
def xor(condition1, condition2):
    return all_of(
        any_of(condition1, condition2),
        any_of(not_(condition1), not_(condition2))
    )
```

Boolean Expressions

```
with show_assert():
    assert_that(
        [1,2, 3],
        xor(
            has_item(1),
            has_item(2)
        )
    )
```

```
Expected: ((a sequence containing <1> or a sequence containing <2>) and (not a sequence containing <1> or not a sequence containing <2>))
     but: (not a sequence containing <1> or not a sequence containing <2>) was <[1, 2, 3]>
```

Floating Point Numbers

```
with show_assert():
    assert_that(
        0.1 + 0.2 - 0.1 - 0.2,
        close_to(1, 0.00001)
    )
```

```
Expected: a numeric value within <1e-05> of <1>
     but: <2.7755575615628914e-17> differed by
<1.0>
```

Strong Checking

```
with show_assert():
    assert_that(
        "hello␣beautifiul␣world",
        string_contains_in_order(
            "hello", "world", "i"
        )
    )

Expected: a string containing 'hello', 'world',
'i' in order
     but: was 'hello beautifiul world'
```

Dictionary

```
with show_assert():
    assert_that(
        dict(value=1),
        has_entry("value", close_to(0.5, 0.3))
    )

Expected: a dictionary containing ['value': a
numeric value within <0.3> of <0.5>]
     but: was <{'value': 1}>
```

What Is a Custom Matcher

Arbitrary condition

What Is a Custom Matcher

Arbitrary condition
Arbitrary description

Primaily Assertion

```
class IsPrime(BaseMatcher):
    def _matches(self, num):
        for factor in range(1, int(num ** 0.5) + 1)
            if num % factor == 0:
                return False
        return True
    def describe_to(self, description):
        description.append(”prime␣number”)

def is_prime():
    return IsPrime()
```

Primaily Assertion

```
with show_assert():
    assert_that(6, is_prime())
```

```
Expected: prime number
     but: was <6>
```

Contains in Order

```
class HasItemsInOrder(BaseMatcher):
    def __init__(self, matchers):
        self.matchers = matchers
```

Matching

```
def _matches(self, sequence):
    things = iter(sequence)
    for matcher in self.matchers:
        for thing in things:
            if matcher.matches(thing):
                break
        else:
            return False
    return True
HasItemsInOrder._matches = _matches
```

Describing

```
def describe_to(self, description):
    description.append_text(
        "a␣sequence␣containing␣"
    )
    for matcher in self.matchers[:-1]:
        description.append_description_of(matcher)
        description.append_text("␣followed␣by␣")
    description.append_description_of(
        self.matchers[-1]
    )
HasItemsInOrder.describe_to = describe_to
```

Wrapping

```
def has_items_in_order(*matchers):
    return HasItemsInOrder(
        [wrap_matcher(matcher)
         for matcher in matchers
        ]
    )
```

Using

```
with show_assert():
    assert_that(
        deque([1, 5, 2]),
        any_of(
            has_items_in_order(1, 3),
            has_items_in_order(2, 1),
        )
    )

Expected: (a sequence containing <1> followed by
<3> or a sequence containing <2> followed by <1>)
     but: was <deque([1, 5, 2])>
```

Checking Output from Code

```
ANSWER = 42
FACTOR = 10

def greet(user):
    print("Greetings,␣", user)
    print(
        "The␣ultimate␣answer␣is",
        ANSWER,
        "as␣you␣might␣know"
    )
    print(
        "Mulitplying␣it␣by",
        FACTOR,
        "you␣get␣the␣important␣concept\n",
        FACTOR * ANSWER
    )
```

Checking Output from Code

```
greet(”pyjamas”)
```

```
Greetings, pyjamas
The ultimate answer is 42 as you might know
Mulitplying it by 10 you get the important concept
 420
```

Checking Output from Code

```
with show_assert():
    with mock.patch("sys.stdout", new=io.StringIO()
        greet("someone")
    output = stdout.getvalue()
    assert_that(
        output,
        string_contains_in_order(
            "Someone",
            "42",
            "420"
        )
    )
```

```
Expected: a string containing 'Someone', '42',
'420' in order
     but: was 'Greetings,  someone\nThe ultimate
answer is 42 as you might know\nMulitplying it by
10 you get the important concept 420\n'
```

Combining Assertions

```
with show_assert():
    assert_that(
        [1, 2, 3],
        any_of(
            has_item(greater_than(3)),
            has_item(less_than(1)),
        )
    )

Expected: (a sequence containing a value greater
than <3> or a sequence containing a value less
than <1>)
     but: was <[1, 2, 3]>
```

Datastructures

```
with show_assert():
    assert_that(
        dict(hello="Greeting", goodbye="Farewell"),
        has_entries(
            hello=ends_with("!"),
            goodbye=not_(ends_with("!")),
        )
    )
```

```
Expected: a dictionary containing {'goodbye': not
a string ending with '!', 'hello': a string ending
with '!'}
     but: value for 'hello' was 'Greeting'
```

Take Aways

Test what is promised

Take Aways

Test what is promised
Do not test what is not

Take Aways

Test what is promised
Do not test what is not
Hamcrest helps you to do that