takathon

Release 0.3.0

Mar 25, 2022

Contents:

A language focused on QA and unit testing.

CHAPTER 1

Features

- Standardized test structure
- Dedicated Python like syntax
- Test domains
- Compatibility with Python
- Support for BDD workflow

CHAPTER 2

Install

```
pip install https://github.com/piwonskp/takathon/archive/0.3.0.tar.gz
```

2.1 Introduction

A test can be separated into three phases:

- **Test preparation** - configuration, environment setup, mocking, creating objects, preparing arguments etc.
- **Invocation** - execution of the code under test
- **Assertion** - a single assertion or series of assertions, often one-liner

Usually test preparation is the most complicated phase. Due to this fact the core point of test is often foggy.

2.1.1 Declarative tests

Pytest does a great job in managing separation between test preparation(fixtures) and actual tests. But there is more to it. Tests are kind of examples which are always up to date - actually that's what doctest do.

So why?

There is a huge amount of informations that can be extracted from tests, not only usage examples. Pytest and doctest are hard to parse both by people and software. That's where declarative tests and domain specific languages come in.

Examples

Consider statement *Function foo takes 3 as an argument and returns 5*. From this simple statement we can deduce that function has type of *Int -> Int*. This process is called type inference.

By analyzing what is being mocked by particular test we can also deduce if the object being tested is a pure function or a procedure. In other words by analyzing test's mocks you can find tested procedure's dependencies.

Tests are a description of code very much like type system is. Tests describe an interface to function, function domain(reasonable arguments that was predicted by author and tested against) and dependencies. While using plain

Python tests this data is basically unavailable. Due to these properties of declarative tests it is preferred to call it specification which function is tested against rather than tests.

2.2 Quickstart

2.2.1 Structure

Basic specification structure:

```
spec:
    [Any operation here applies to all test cases below]
    domain <arguments>:
        [Operations declared here apply to this particular test case]
        results <expected result>
```

Let’s see what all of this means:

- **spec** - indicates start of specification.
- **domain** - indicates single test case. Domain statement is followed by arguments passed to the tested function.
- **results** - specifies the expected result of test case.

2.2.2 Ordinal data

Tests shall be simple. As simple as saying “if I provide you with arguments X I expect the result Y”. So let’s try to implement function that provided with negative value returns 0. Otherwise the function returns its input.

```
def flatten_positive(x):
    """
    spec:
        domain 32: results 32
        domain 0: results 0
        domain interval (-Infinity, 0): results 0

        # Floats are also allowed
        domain 67.6: results 67.6
        domain interval (-Infinity, 0): results 0.0
    """
    if x < 0:
        return 0
    return x
```

Test cases can include plain values or sets of values as arguments. That is why the keyword is called *domain* rather than *arguments*. *interval* is used to specify input ranges in which function has constant output. Additionally *Infinity* is a special value that indicates no upper boundary of range (*-Infinity* means no lower boundary).

Warning: Interval is mechanism of specyfing tests declaratively. Under the hood argument contained within specified range is randomly generated. The language **does not** prove the statement is true for all possible combinations of arguments by any means.

2.2.3 Nominal data

For unordered test data you can use *any_of* to indicate that the result is constant for specified objects.

```
from enum import Enum

Color = Enum("Color", ("BLUE", "CYAN", "GREEN", "YELLOW", "RED"))


def is_red(color):
    """
    spec:
        domain any_of(Color.BLUE, Color.CYAN, Color.GREEN, Color.YELLOW):
            results False
        domain Color.RED: results True
    """
    return color == Color.RED
```

2.2.4 Running tests

To run tests use command:

```
takathon <path-to-file-or-directory>
```

Additionaly if you want more detailed output you can use:

```
takathon -v info <path-to-file-or-directory>
```

2.2.5 Python features

Plain Python imports works out of the box within the specification so you can bring any Python object into the test scope. No need to bloat the module itself. Python style comments are also supported.

2.3 Documentation

Apart from comments in specification there are two additional keywords which are preferred to document the object and describe test cases. These are:

- title
- description

2.3.1 Short summary

title is used to present test results to the user.

When used on specification level *title* is a short summary of object's goal.

title on test case level indicates test case goal.

When *title* is missing user will be presented with function name(specification level) or domain(test case level).

2.3.2 Long description

description contains actual documentation for programmers.

When used on specification level *description* is a documentation for object itself.

description on test case level helps understand reasons of particular test and its quirks.

2.3.3 Examples

Specification level documentation

Note that example below is syntactically correct and will compile without any error. You can preserve TDD workflow without any additional action or jumping between files. You also are not required to have any test cases in specification.

```
def integral(data):
    """
    spec:
        title: Calculate integral
        description:
            Calculates integral using Monte Carlo
    """
```

Test case level documentation

```
def identity(obj):
    """
    spec:
        domain 1:
            title: Should work properly for integers
            description: Provided with 1 function returns 1
            results 1
    """
    return obj
```

2.4 Mocks

2.4.1 Mocking within the same module

To indicate that mocked object is in the same module as procedure being tested prepend the name of it with dot.

```
def increment_input():
    """
    spec:
        title: Incrementation of input
        from unittest.mock import MagicMock
        domain:
            title: Should return 4 when input is 3
            mock .input as MagicMock(return_value='3')
            results 4
    """
    return int(input()) + 1
```

2.4.2 Mocking external module

Let's assume you want to reuse procedure defined in previous paragraph. To do that you have to specify absolute path to *input*. In this case mocked procedure is contained within *examples.mocks.mock_input* module. Note that you don't have to import *Mock* to scope since it is already builtin.

```
from examples.mocks.mock_input import increment_input
from examples.result_validation.factorial import factorial


def mock_external():
    """
    spec:
        mock examples.mocks.mock_input.input as Mock(return_value=1)
        domain : results 2
        mock examples.mocks.mock_input.input as Mock(return_value=5)
        domain : results 720
    """
    value = increment_input()
    return factorial(value)
```

2.5 Result validation

Apart from standard result validation there are also other methods of validating correctness of function call.

2.5.1 Throws

If function is expected to raise an exception for particular input you can use *throws* statement.

```
def factorial(n):
    """
    spec:
        title: Factorial
        domain -1:
            title: Should be incalculable for negative values
            description:
                Function raises proper exception
                when called with negative value
            throws ValueError('factorial() not defined for negative values')
        domain 0:
            title: Should return 1 on boundary
            description: Function should return 0 when called with 0
            results 1
        domain 1: results 1
        domain 3: results 6
        domain 4: results 24
        domain 5: results 120
    """
    if n < 0:
        raise ValueError("factorial() not defined for negative values")

    if n == 0:
        return 1
    return n * factorial(n - 1)
```

2.5.2 Between

If your function returns inacurrate results you can use *between* to validate if result is in desired range.

```
def square_derivative(x):
    """
    spec:
        domain 10:
            result between (19.9, 20.1)
    """
    eps = 0.01
    f = lambda x: x * x
    return (f(x + eps) - f(x)) / eps
```

2.5.3 Validate by function

The most generic way of validating result. At first you have to define predicate in plain Python. The predicate is of type *Result -> Bool* ie. takes the result of function call and returns boolean. The boolean value indicates whether test passed or not.

```
def got_name(result):
    return result.get("name")
```

You can either define the predicate in the same file or import it for the sake of test/code separation as shown below. To validate result using function use *result <foo_name>* statement.

```
def check_by_function():
    """
    spec:
        from examples.result_validation.check_by_function.test_utils import got_name
        domain:
            result got_name
    """
    return {"name": "example", "value": 1}
```

2.6 OOP features

The language supports object oriented programming features such as creating objects. The features provide the way to perform basic method testing. Advanced test scenarios for classes are not supported yet.

2.6.1 Static methods

Static methods can be tested like any other procedure.

```
class StaticMethods:
    x = 5
    y = 1

    def __init__(self, a, b):
        self.a = a
        self.b = b

```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def sub(a, b):
        """
        spec:
            domain 10, 7: results 3
            # Check if creation of an object changes the result
            new 40, 32
            domain 10, 7: results 3
        """
        return a - b
```

2.6.2 Methods

To create an object *new* keyword is used. *new* keyword creates the object by passing arguments to the class constructor. It provides a way to realize the test scenario:

```
new_object = ClassName(constructor_arguments)
result = new_object.method(method_arguments)
assert result == expected_result
```

This test scenario allows to test methods as separate units independently of other methods of the class.

```
class SingleMethodWithoutDocs:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self, c):
        """
        spec:
            # Test creation of object
            new 10, 20
            domain 2: results 32
            # Make sure object is replaced with a new one
            new -6, 2
            domain 4: results 0
        """
        return self.a + self.b + c

    def sub(self):
        """
        spec:
            new 20, 10
            domain : results 10
            new 40, 32
            domain : results 8
        """
        return self.a - self.b

    def mul(self):
        return self.a * self.b
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search