Python Testing Frameworks: a summary

Testing is a crucial part of software development, ensuring code reliability and functionality. Python offers a wide selection of testing frameworks catering to different needs and complexities.

This handout provides a concise summary of popular Python testing frameworks, organized by their level of expertise required, along with their primary strengths, main advantages, and usage examples.

Easy Level Frameworks

These are frameworks that are easy to use and can be incorporated very quickly into your code base. They are limited in their scope.

doctest

- **Level of Difficulty:** Easy
- **Primary Strengths:** Testing code examples within docstrings.
- **Pros:**
    - Part of the Python standard library.
    - Simple to use; no additional setup required.
    - Great for testing small functions and ensuring documentation accuracy.

Example Usage:

```
def add(a, b):
    """
    Adds two numbers.

    >>> add(2, 3)
    5
    >>> add(-1, 1)
    0
    """
    return a + b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

pytest

- **Level of Difficulty:** Easy
- **Primary Strengths:** Simplifies writing small tests with minimal code.
- **Pros:**
    - Simple and intuitive syntax.
    - Detailed assertions and informative error messages.
    - Extensive plugin ecosystem for extended functionality.
    - Supports fixtures and parameterized testing.

Example Usage:

```
# test_math.py

def add(a, b):
    return a + b

def test_add_positive_numbers():
    assert add(2, 3) == 5

def test_add_negative_numbers():
    assert add(-1, -1) == -2
```

Run tests with the command:

```
pytest test_math.py
```

Moderate Level Frameworks

These are frameworks that are more complex to use but do offer a more organized approach to testing in python.

unittest

- **Level of Difficulty:** Moderate
- **Primary Strengths:** Provides a solid foundation for organizing tests using classes.
- **Pros:**
    - Part of the Python standard library.
    - Supports test cases, suites, and fixtures.
    - Familiar xUnit style, similar to JUnit for Java.
    - Detailed test reports and assertions.

Example Usage:

```
import unittest

def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):

    def test_add_positive_numbers(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(add(-1, -1), -2)

if __name__ == '__main__':
    unittest.main()
```

Hypothesis

- **Level of Difficulty:** Moderate
- **Primary Strengths:** Property-based testing; generates test cases automatically.
- **Pros:**
    - Discovers edge cases that traditional tests might miss.
    - Reduces the effort in writing multiple test cases manually.
    - Can be combined with other testing frameworks like **unittest** or **pytest**.

Example Usage:

```
from hypothesis import given
import hypothesis.strategies as st

def add(a, b):
    return a + b

@given(st.integers(), st.integers())
def test_add(a, b):
    assert add(a, b) == a + b
```

Run tests with the command:

```
pytest test_math.py
```

Advanced Level Frameworks

These are more industrial and serious frameworks that help in automating testing suites and test cases.

Robot Framework

- **Level of Difficulty:** Advanced
- **Primary Strengths:** Keyword-driven testing suitable for acceptance testing.
- **Pros:**
    - Enables writing readable and understandable tests for non-programmers.
    - Supports data-driven testing and can integrate with Selenium for web testing.
    - Provides detailed logs and reports.

Example Usage:

```
*** Test Cases ***
Addition Test
    ${result}=    Add Numbers    2    3
    Should Be Equal    ${result}    5

*** Keywords ***
Add Numbers
    [Arguments]    ${a}    ${b}
    ${sum}=    Evaluate    ${a} + ${b}
    [Return]    ${sum}
```

Run tests with the command:

```
robot test_math.robot
```

behave

- **Level of Difficulty:** Advanced
- **Primary Strengths:** Supports Behavior-Driven Development (BDD) with Gherkin syntax.
- **Pros:**
    - Facilitates collaboration between technical and non-technical team members.
    - Allows writing test scenarios in plain English.
    - Integrates well with Selenium for web application testing.

Example Usage:

First we create a *Feature* file: *(**add.feature**)*

```
Feature: Addition

  Scenario: Add two positive numbers
    Given I have numbers 2 and 3
    When I add them
    Then the result should be 5
```

Then we create the step definitions in python: ***(steps/add_steps.py)***

```
from behave import given, when, then

def add(a, b):
    return a + b

@given('I have numbers {a:d} and {b:d}')
def step_given_numbers(context, a, b):
    context.a = a
    context.b = b

@when('I add them')
def step_when_add(context):
    context.result = add(context.a, context.b)

@then('the result should be {expected_result:d}')
def step_then_result(context, expected_result):
    assert context.result == expected_result
```

Run tests with the command:

```
pytest test_math.py
```

Conclusion

As it is with all things in life, selecting the right testing framework **depends on** the project's **requirements** and the team's **familiarity with the tools**.

If you require **quick and simple tests** then **doctest** and **pytest** are excellent choices, as they are easy to use with minimal setup needed.

If you need more structured testing then **unittest** and **Hypothesis** provide robust features.

For teams adopting acceptance testing or BDD practices, **Robot Framework** and **behave** offer very powerful and versatile capabilities.

References

- Python Documentation - doctest
- pytest Official Site
- Python Documentation - unittest
- Hypothesis Official Site
- Robot Framework Official Site
- behave Documentation