Hey there,

Thanks for downloading this Free Pytest Mastery and Best Practices Guide.

Here's a handy, bullet-point list of the **Top Pytest and Unit Testing Best Practices** derived from over 9 years of experience building production applications as a software developer and data engineer, and reading countless books on the topics.

I've also included example code to better illustrate the concepts.

Refer to this guide regularly to help you write tests that are easy to maintain, reliable, and fast.

Good luck and happy testing!

Pytest and Unit Testing Best Practices

Tests Should Not Include Implementation Details

This is the most important one. Your tests should not include implementation details. This means that you should avoid testing private methods, properties, or any other internal details of the code. You want to test WHAT the code does, not HOW it does it.

Example Code:

```
# Bad
def test_add_numbers():
    assert 2+2 == 4  # Includes implementation details

# Good
def test_add_numbers():
    assert add_numbers(2, 2) == 4 # Doesn't include
        implementation details
```

Tests Should Be Fast

Keep your tests fast and agile by avoiding slow and unnecessary operations like sleep, repeated setup and teardown (use Fixtures instead). This will help you run your tests more frequently and get faster feedback while not being too much of a development bottleneck.

Example Code:

```
# Bad
def test_add_numbers():
    time.sleep(10)  # Other slow operations like database calls,
        network calls, test data set up for each test etc.
    assert add_numbers(2, 2) == 4

# Good
def test_add_numbers():
    assert add_numbers(2, 2) == 4
```

Tests Should Be Independent

Each test should be independent of the other. This means that the outcome of one test should not affect the outcome of another test and ideally shouldn't share states. This is important because it helps you identify the exact cause of a test failure and makes your tests more reliable.

This means that you should avoid using global variables, databases, or any other shared resources that can be modified by one test and affect the outcome of another test. Resetting these resources using setup and teardown methods can help.

Example Code:

```
# Bad
result = 0

def test_add_numbers():
    global result
    result = add(1, 2)
```

```
    assert result == 3

def test_subtract_numbers():
    assert subtract(result, 1) == 2


# Good
def test_add_numbers():
    assert add(1, 2) == 3

def test_subtract_numbers():
    assert subtract(2, 1) == 1
```

Each Test Should Test One Thing

Each test should test one thing and one thing only. Avoid testing multiple things in one test method as this makes debugging difficult and can lead to false positives.

Example Code:

```
# Bad
def test_numbers():  # Tests 2 things in one test making it
        harder to debug
    assert add(1, 2) == 3
    assert subtract(2, 1) == 1

# Good
def test_add_numbers(): # Tests one thing
    assert add(1, 2) == 3

def test_subtract_numbers(): # Tests one thing
    assert subtract(2, 1) == 1
```

Tests Should Be Readable

Write tests that are easy to read and understand. Use good naming conventions and avoid abbreviations. This will help you and your team understand the purpose of the test and the expected outcome.

Example Code:

```
# Bad
def test_1():
    assert add(1, 2) == 3
```

```
# Good
def test_add_numbers():
    assert add(1, 2) == 3
```

Tests Should Be Deterministic

A test should always produce the same result given the same input. This means that you should avoid using random data or any other non-deterministic operations in your tests.

An exception is when implementing controlled Property-Based Testing or Pytest Parametrization.

Example Code:

```
# Bad
def test_add_numbers():
    a = random.randint(1, 10)  # Non-deterministic
    b = random.randint(1, 10)  # Non-deterministic
    assert add(a, b) == a + b  # includes implementation details

# Good
def test_add_numbers():
    assert add(1, 2) == 3
```

Use Single Assert Per Test Method

Each test method should have a single assert statement. This makes it easier to identify the exact cause of a test failure. If you have multiple asserts in a test method, you won't know which one failed.

Example Code:

```
# Bad
def test_add_numbers():
    result = add(1, 2)
    assert result == 3
    assert result + 1 == 4

# Good
def test_add_numbers():
    result = add(1, 2)
    assert result == 3
```

There are cases where you might want to use multiple asserts in a test method, but this should be the exception rather than the rule. For example, when testing a data structure or testing API response codes and data, this is fine.

```
# Good
def test_api_response():
    response = requests.get("https://api.com/data")
    assert response.status_code == 200
    assert response.json() == {"data": "some data"}

def test_data_structure():
    data = {"name": "John", "age": 30}
    assert data["name"] == "John"
    assert data["age"] == 30
```

Use Dummy Data and Empty Databases in Testing

Use dummy data and empty databases when testing. This will help you avoid side effects and ensure that your tests are deterministic and independent while representing real-world scenarios. You can leverage Fixture Scopes to generate dummy data or setup and teardown test databases.

Example Code:

```
# Bad
def test_db_operations(): # Connects to REAL database
    # Connect to REAL database
    # Insert data into the database
    # Modifies data in the database
    # Deletes data from the database
    # Disconnect from the database
```

The above risks affecting the real database and is very dangerous as it can result in data loss.

```
# Good
def test_db_operations(db):
        # Uses a dummy database (in-memory or temporary)
    # Insert data into the database
    # Modifies data in the database
    # Assert the outcome
    # Deletes data from the database

        # Tear down the database or use a fixture scope to manage
        the database across tests
```

Use Dependency Injection

Dependency Injection (DI) is a design pattern allowing for better modularity and making code more testable, maintainable, and extensible. It also enhances test isolation and flexibility.

It can simplify the process of providing a class or function with the external objects it depends on, rather than having it construct them internally.

This is particularly useful in scenarios where the components being used can vary (e.g., during testing, you might want to inject mock objects instead of real ones).

Example Code:

```
# Bad - Without Dependency Injection
class EmailService:
    def send_email(self, message):
        print(f"Sending email: {message}")
        # Imagine this method actually sends an email

class Mailer:
    def __init__(self):
        self.email_service = EmailService() # Directly
        instantiates EmailService

    def send_message(self, message):
        self.email_service.send_email(message)
```

In this setup, `Mailer` is directly dependent on `EmailService`, making it difficult to test `Mailer` without also involving the actual sending of emails.

```
# Good - With Dependency Injection
class EmailService:
    def send_email(self, message):
        print(f"Sending email: {message}")
        # Actual email sending logic

class Mailer:
    def __init__(self, email_service):
        self.email_service = email_service

    def send_message(self, message):
        self.email_service.send_email(message)

# Test
class MockEmailService:
```

```
    def send_email(self, message):
        print(f"Mock send email: {message}")
        # Mock email sending logic for testing

# For production use
real_email_service = EmailService()
mailer = Mailer(real_email_service)
mailer.send_message("Hello, this is a real email!")

# For testing
mock_email_service = MockEmailService()
test_mailer = Mailer(mock_email_service)
test_mailer.send_message("Hello, this is a test email!")
```

In the above example, `Mailer` is no longer directly dependent on `EmailService`, making it easier to test `Mailer` without involving the actual sending of emails.

Use Setup and Teardown To Manage Test Dependencies

Use setup and teardown methods to manage test dependencies.

This will help you avoid code duplication and make your tests more maintainable. You can make use of the `yield` keyword and fixture scopes to manage setup and teardown in Pytest.

Example Code:

```
# Bad
def test_db_write_():
    # Setup
    db = connect_to_db()
    # Test
    db.write(data)
    assert db.read() == data
    # Teardown
    db.disconnect()

def test_db_read():
    # Setup
    db = connect_to_db()
    # Test
    assert db.read() == data
```

```
    # Teardown
    db.disconnect()

# Good
@pytest.fixture
def db():
    db = connect_to_db()
    yield db
    db.disconnect()

def test_db_read(db):
    assert db.read() == data
```

Organize Tests Effectively (Group related tests together)

Group related tests together and organize them effectively. This will help you find and run tests easily and make your tests more maintainable. You can use test classes, modules, and packages to organize your tests.

Example Code:

```
project/
│
├── app/
│   └── calculator.py
│
└── tests/
    ├── unit/
    │   ├── test_calculator.py
    │   └── ...
    ├── integration/
    │   ├── test_db_operations.py
    │   └── ...
    └── ...
```

In the above example, we've clubbed `unit` and `integration` tests into their own directory.

You can also use Pytest Markers to run specific tests or groups of tests.

```
pytest -m "unit"
```

Mock and Patch External Dependencies

When working with external dependencies for example - databases, Rest APIs, or any other external services, you should mock or patch these dependencies. This will help you isolate the code under test and make your tests more reliable and shielded from external changes that could break your tests.

Mocking refers to the process of replacing a real object with a fake object that simulates the behavior of the real object. Patching refers to the process of replacing an object in a module with a mock object.

Example Code:

```
# Bad
def test_get_data_external_api():
    response = requests.get("https://REAL_API.com/data")  # This
        will make a real network call and fail if the API is
        unavailable or endpoint changes
    assert response.status_code == 200
    assert response.json() == {"data": "some data"}

# Good
def test_get_data_external_api(mocker):
    # Mock
    mock_response = mocker.Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {"data": "some data"}

    # Patch
    mocker.patch("requests.get", return_value=mock_response)
    response = requests.get("https://REAL_API.com/data")
    assert response.status_code == 200
    assert response.json() == {"data": "some data"}
```

Practice Test Driven Development (TDD) where possible

Practicing Test Driven Development (TDD) means that you should write tests before you write the code using the **Red-Green-Refactor** Method. This will help you think about the design of your code and ensure that your code is testable, while helping you make sure existing functionality is not broken when adding new features.

Regularly Review and Refactor Tests

Regularly reviewing and refactoring tests helps you identify and remove any unnecessary or redundant tests.

It will also help you ensure that your tests are always in a working state and that they are easy to understand and maintain.

Use Test Coverage But Don’t Prioritize It

Use test coverage to identify areas of your code that are not covered by tests. However, don’t prioritize test coverage over writing good tests. Test coverage only shows the coverage % of the code you’ve written so - if your code and specs are wrong, you’ll have 100% coverage of the wrong code.

Test for Security Issues as Part of Your Unit Tests

Test for security issues as part of your unit tests. This means that you should test for common security vulnerabilities like SQL Injection, Cross-Site Scripting, and others as part of your unit tests. This will help you catch security issues early and ensure that your code is secure. You can use libraries like `bandit` and `safety` to help with this.

Tests Should Be Part Of The CI/CD Process

Your tests should be part of the CI/CD process. This means that they should be run automatically every time you push code to the repository and it’s deployed to a server. This will help you catch bugs early and ensure that your code is always in a working state.

You can use tools like GitHub Actions, CircleCI, Jenkins, and others to run your tests as part of a CI/CD pipeline.

Write Behavior-Driven Tests (BDD) where possible

Write behavior-driven tests that describe the behavior of the code from the user’s perspective. This is very useful for testing features and behaviors from the user’s perspective for code like APIs, UI components, and others. You can use libraries like `pytest-bdd` and `behave` to help with this.

Manage Configuration via Config Files

Manage configuration via config files instead of hardcoding configuration values in your. It keeps the execution repeatable and makes it easier to manage different configurations for different environments. Pytest support

config files like `pytest.ini` , `pyproject.toml` , `setup.cfg` and `tox.ini` for managing configurations.

Example Code:

```
# pytest.ini
[pytest]
[pytest]
minversion = 6.0
testpaths =
    tests
    integration
addopts = -ra -q
env =
    ENV=dev
    DB_URL=sqlite:///:memory:
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    fast: marks tests as fast
```

Reduce or Limit Network Access

Reduce or limit network access in your tests. This will help you avoid flaky tests and ensure that your tests are fast and reliable. You can use packages like `pytest-socket` help with this.

Use Temporary Files and Directories for Testing

Using temporary files and directories for testing help avoid side effects and ensures that your tests don't leave any stale data around leading to future test contamination.

You can use the Pytest built-in `tmp_path` or `tmpdir` fixtures to help with this, defined by the correct scope.

I hope you found this guide helpful, please share it with your friends and colleagues whom you think might benefit from it.

Visit Pytest with Eric for more in-depth tutorials and resources on Pytest and Unit Testing.