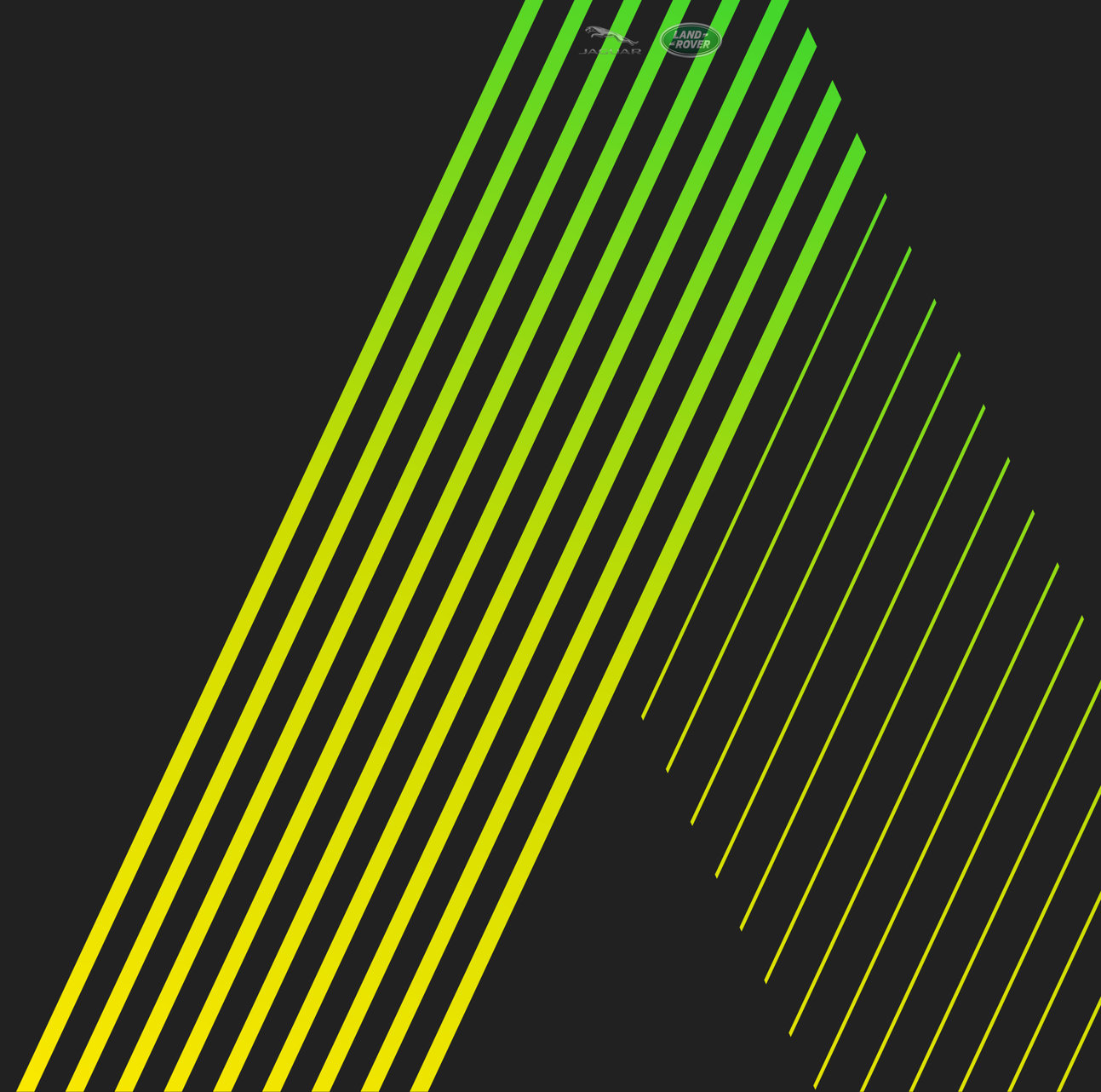
Unit Testing 1



Overview

- Introduction to Unit Testing
- Why and How We Unit Test
- Testing Pathways
- Test Cases
- Development and Testing
- Testing frameworks

Learning Objectives

- Define unit testing
- Identify testing pathways
- Explore some test cases
- Compare TDD (test driven development) and Non-TDD
- Create a simple unit test
- Getting started with `pytest`

What is a Unit?

A "unit" of code is considered to be the smallest testable chunk of software which performs a very specific job / task.

```
def add_two_numbers(a, b):
    return a + b
```

What is Unit Testing?

Unit Testing is then the process of executing this unit of code in isolation under certain conditions or scenarios to test its behaviour.

```
add_two_numbers(1, 1) # Expected 2
add_two_numbers(-1, 0) # Expected -1
add_two_numbers(1.00234, 0.3456) # Expected 1.34794
add_two_numbers("test", 1) # Expected Error
add_two_numbers() # Expected Error
```

Testing Pathways

When testing we refer to our test scenarios as following two distinct paths:

- The Happy Path
- The Unhappy Path

Happy Path

We test successful scenarios

```
add_two_numbers(1, 1)
add_two_numbers(0, -10)
add_two_numbers(52130032132321321, 0.000000022330)
```

Unhappy Path

We test unsuccessful scenarios

```
add_two_numbers("test", 1)
add_two_numbers(1)
add_two_numbers()
```

Test Cases

We can also define certain *test cases* when we test:

- Common Case
- Edge Case
- Corner Case

Common Case

This occurs at normal operating parameters

```
add_two_numbers(100, 100)
```

Edge Case

This occurs at the extreme min / max parameter envelope

```
add_two_numbers(0, 10**10000)
```

Corner Case

This occurs outside of normal operating parameters

```
add_two_numbers("text", 10**10000)
```

Why do we care?

- A good testing strategy outlines the operational envelope of our software.
- Failing tests indicate where we need to improve our software.
- Passing tests are an indicator of software quality and robustness.

```
robust_software == happy_users == happy_employer
```

Writing our first test

The Three A's:

1. Create Test Data (Assemble)
2. Execute the unit we're testing and pass in the test data (Act)
3. Verify the result matches our expectations (Assert)

```python
def test_adds_two_numbers():
    # Assemble
    a = 7
    b = 12
    expected = 19

    # Act
    result = add_two_numbers(a, b)

    # Assert
    assert result == expected

test_adds_two_numbers()
```

Techniques for writing unit tests

Just as we write application code, we write test code in much the same way.

There are however two main approaches to writing unit-tests:

- Write the code then the tests (non TDD)
- Write the tests then the code (TDD)

Non Test Driven Development

1. Read, understand, and process the feature or bug request.
2. Implement the code that fulfils the requirement.
3. Test the code works by writing a unit test.
4. Clean up your code by refactoring.
5. Rinse, lather and repeat.

Example

1. Write the code and hope it works:

```
def add_two_numbers(a, b):
    return a + b
```

2. Write the test and hope it passes:

```
def test_add_two_numbers(a, b):
    expected = 10
    actual = add_two_numbers(5, 5)
    assert expected == actual
```

3. Fix the code if the tests fails

Test Driven Development (TDD)

1. Read, understand, and process the feature or bug request.
2. Translate the requirement by writing a unit test.
3. Write the minimum amount of code to get the test to pass.
4. Rinse, lather and repeat.

Step 1 - Implement the minimal amount of code needed to pass the test

```python
def add_two_numbers(a, b):
    return 10

def test_add_two_numbers():
    # Arrange
    a = 5
    b = 5
    expected = 10

    # Act
    actual = add_two_numbers(a, b)

    # Assert - pass
    assert expected == actual

test_add_two_numbers()
```

Step 2 - Fully implement function, get test to pass

```python
def add_two_numbers(a, b):
    return a + b

def test_add_two_numbers():
    # Arrange
    a = 5
    b = 5
    expected = 10

    # Act
    actual = add_two_numbers(a, b)

    # Assert - pass
    assert expected == actual

test_add_two_numbers()
```

Benefits of TDD

- Gets you into the dependency injection mindset, which will help your code to be more rigorous.
- Requires you to implement just enough code and prevents you predicting the future. It ensures requirements are understood and met explicitly. It saves time and improves velocity.
- Once you get a test to pass, you know that any refactoring of the code needs to work such that the test still passes. If it doesn't you've either implemented your functionality wrong, or the test was written incorrectly.

Example [code-along] - part 1

We have been asked to write a python function named `price_updater` with the following requirements:

1. It should receive 2 arguments, prices (list[float]) and increase rate (float) and return prices list with the same order and all values increased by the rate.
2. If data type of for any of the prices inside price list was not `float`, return `Incorrect Price Detected!` as a string.
3. Constraints:
   - 0 <= price <= 100,000
   - 0 <= increase_factor <= 1

Exercise 1

Instructor to distribute exercise.

Testing frameworks - `pytest` & `unittest`

- Provides a framework upon which to write and run our tests
- Includes helper objects and functions for versatile mocking, and spying
- Provides a test-runner for test detection and verbose results
- Includes additional assertions for diverse testing scenarios

Installing `pytest`

You can install it globally with:

```
$ pip install pytest
```

Alternatively, you can add it to your `requirements.txt` inside your virtual environment.

Running `pytest`

1. File names should begin or end with `test`, as in `test_example.py` or `example_test.py`.
2. Function names should begin with `test_`. So for instance: `test_example`.
3. If tests are defined as methods on a class, the class should start with `Test`, as in `TestExample`.
4. You can run `pytest --collect-only` to see which tests `pytest` will discover, without running them.

Example 1

```python
# test_additions.py
def add_two_numbers(a, b):
    return a + b

def test_add_two_numbers():
    expected = 5
    actual = add_two_numbers(4, 1)
    assert expected == actual
```

Copy the code to a Python file, run `python -m pytest` or just `pytest` and watch the output. Hopefully you should see some information about 1 test passing.

To increase verbosity and see print results in the terminal add `-v -s` flags to the previous command.

Example 2

Test exception:

```
# test_additions.py
import pytest


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

def test_exception_for_non_numeric_args():
    with pytest.raises(Exception):
        add_two_numbers('a', 10)
```

Example [code-along] - part 2

Add the following requirements to `price_updater` function:

[New Requirements]

1. If value of increase rate had non-numeric data type, it should throw `TypeError`.
2. If value of increase rate was outside the defined constraint it should throw `ValueError`.

Exercise 2

Instructor to distribute exercise.

Learning Objectives Revisited

- Define unit testing
- Identify testing pathways
- Explore some test cases
- Compare TDD (test driven development) and Non-TDD
- Create a simple unit test

Terms and Definitions Recap

- `Unit`: The smallest testable chunk of code.
- `TDD`: Test Driven Development. The process of writing tests first.
- `Happy Path`: Successful test scenarios.
- `Unhappy Path`: Unsuccessful test scenarios.
- `Corner Case`: Outside normal parameters.
- `Edge Case`: Extreme min/max parameters.

Further Reading

- Unit Testing: Best Practices
- Pytest: Beginner Guide