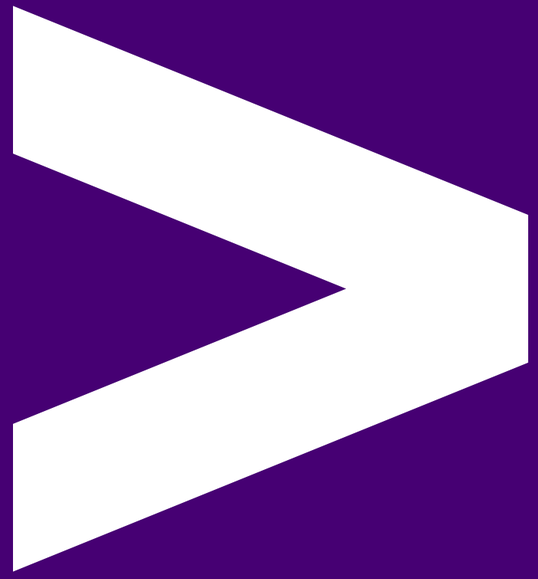
School of Tech
part of accenture

Unit Testing 1



School of Tech
part of accenture

Overview

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

School of Tech
part of accenture

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
```

School of Tech
part of accenture

Testing Pathways

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

- The Happy Path
- The Unhappy Path

Happy Path

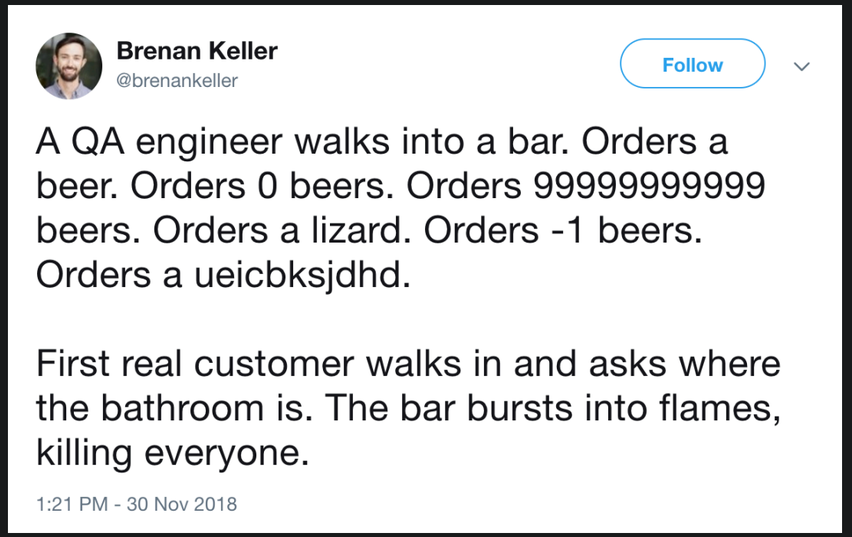
We test successful scenarios:

```
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()
```



Brenan Keller
@brenankeller

Follow

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM - 30 Nov 2018

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)
```

Emoji Check:

Do you feel you understand a bit more about what unit testing is? Say so if not!

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

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
```

School of Tech
part of accenture

Writing our first test

The Three A's:

Arrange
Create Test Data.

Act
Execute the unit we're testing and pass in the test data.

Assert
Verify the result matches our expectations.

```python
def test_add_two_numbers():
    # Arrange
    a = 7
    b = 12
    expected = 19

    # Act
    result = add_two_numbers(a, b)

    # Assert
    assert result == expected

test_add_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():
    expected = 10
    actual = add_two_numbers(5, 5)
    assert expected == actual
```

3. Fix the code if the tests fails

Improving assertion output

We also have the option to display a message containing debug information when our assertion fails:

```
def test_add_two_numbers():
    expected = 8
    actual = add_two_numbers(5, 5)
    assert expected == actual, \
        f"expected '{expected}' but got '{actual}'"
```

This outputs `AssertionError: expected '8' but got '10'`.

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

```
# additions.py
def add_two_numbers(a, b):
    return 10
```

School of Tech
part of accenture

Step 2 - Run the test

```python
# test_add_two_numbers.py
from additions import add_two_numbers

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 3 - Fully implement function, get test to pass

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

School of Tech
part of accenture

Step 4 - Run the test

```python
# test_add_two_numbers.py
from additions import add_two_numbers

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

School of Tech
part of accenture

Emoji Check:

Do you feel you understand what TDD is? Say so if not!

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

Exercise prep

> Instructor to give out the zip file of exercises for `unit-testing-1`
>
> Everyone please unzip the file

Exercise time

> From the zip, you should have a file `exercises/unit-testing-1-exercise-1.md`
>
> Let's all do the exercises included in this file

Emoji Check:

How did the exercises go? Are unit tests making more sense now?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

School of Tech
part of accenture

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:

```
# Mac/Unix
$ python3 -m pip install pytest
# or on Windows
$ py -m pip install pytest
```

To check pytest is installed correctly, run the command `pytest --version`.
You should see an output like "pytest 7.4.3"

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` with `--collect-only` to see which tests `pytest` will discover, without running them.
5. Similar to pip, always add `python3 -m` (MacOS / Unix) `py -m` (Windows) at the start of the command when running `pytest`, e.g. `python3 -m pytest` (MacOS / Unix) `py -m pytest` (Windows)

Example 1

```python
# additions.py
def add_two_numbers(a, b):
    print('The function started...')
    return a + b

# test_additions.py
from additions import add_two_numbers

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

Example 1 continued

- Copy the code to a python file called `additions.py`
- Copy the testing code to a python file called `test_additions.py`
- In your terminal, run `python3 -m pytest` (MacOS / Unix) or `py -m pytest` (Windows)
- Watch the output

School of Tech
part of accenture

Example 1 continued

- Hopefully you should see some information about 1 test passing!

```
test/test_etl_functions.py::test_load_csv_is_successful PASSED                                                [  8%]
test/test_etl_functions.py::test_load_csv_will_throw_error_when_filename_is_invalid PASSED                    [ 16%]
test/test_etl_functions.py::test_load_csv_removes_customer_name_successfully PASSED                           [ 25%]
test/test_etl_functions.py::test_load_csv_removes_card_number_successfully PASSED                             [ 33%]
test/test_etl_functions.py::test_separate_order_string_to_items_dict_is_successful PASSED                     [ 41%]
test/test_etl_functions.py::test_separate_order_string_to_items_dict_increments_quantity_successfully PASSED  [ 50%]
test/test_etl_functions.py::test_split_date_time_to_sql_is_successful PASSED                                  [ 58%]
test/test_etl_functions.py::test_get_location_entry_is_successful PASSED                                      [ 66%]
test/test_etl_functions.py::test_get_all_items_is_successful PASSED                                           [ 75%]
test/test_etl_functions.py::test_get_all_items_returns_correct_number_of_items PASSED                         [ 83%]
test/test_etl_functions.py::test_get_orders_and_mappings_is_successful PASSED                                 [ 91%]
test/test_etl_functions.py::test_get_orders_and_mappings_return_correct_length_of_orders_and_mappings PASSED [100%]
```

Example 1 continued

- Add `-v -s` flags to the command: `python3 -m pytest -v -s` (MacOS / Unix) `py -m pytest -v -s` (Windows)
- Notice anything different?

`pytest` command line flags

- `-v`: Increases the verbosity to list results for all tests
- `-s`: Instructs pytest to display any terminal output for all tests (usually it swallows it)

Notes on running with Pytest

> When we started the session we added calls to our test functions in the test file directly.
>
> When using pytest, we do **not** [need to] do this, as pytest runs the files for us!

Example 2 - Testing Exceptions

We can test any exceptions our code throws by using the `pytest.raises()` method, like so:

```
# test_additions.py
import pytest

from additions import add_two_numbers

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

Example 2 - Testing exceptions continued

In order to be stricter (and so more accurate) in our tests, we can also be more specific about the exception type, and test the message by giving a RegEx to match the message produced:

```
# test_additions.py
import pytest
from additions import add_two_numbers

def test_exception_for_non_numeric_args():
    with pytest.raises(ValueError, match=r'not a number'):
        add_two_numbers('a', 10)
```

See pytest how-to/assert for more.

Example [code-along] - part 2

Add the following requirements to the `price_updater` function:

[New Requirements]

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

Exercise time

> From the `unit-testing-1` zip, you should have a file `exercises/unit-testing-1-exercise-2.md`
>
> Let's all do the exercises included in this file

Emoji Check:

How did the exercises go? Is the `pytest` framework making more sense now?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

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.

School of Tech
part of accenture

Overview - recap

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

School of Tech
part of accenture

Learning Objectives - recap

- 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`

Further Reading

- Unit Testing: Best Practices
- Pytest: Beginner Guide

School of Tech
part of accenture

Emoji Check:

On a high level, do you think you understand the main concepts of this session? Say so if not!

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively