(/)

Curriculum
Short Specializations
Average: **54.31%**

You just released the advanced tasks of this project. Have fun!

0x03. Unittests and Integration Tests

UnitTests **Back-end** **Integration tests**

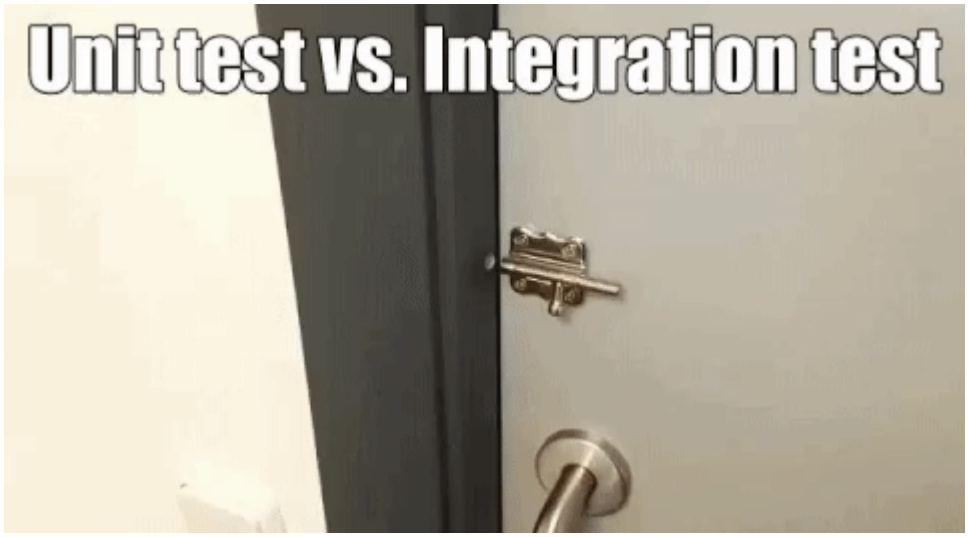
Weight: 1

Project over - took place from Jul 25, 2024 3:00 AM to Jul 30, 2024 3:00 AM

An auto review will be launched at the deadline

In a nutshell…

- **Auto QA review:** 0.0/26 mandatory & 0.0/4 optional
- **Altogether: 0.0%**
  - Mandatory: 0.0%
  - Optional: 0.0%
  - Calculation: 0.0% + (0.0% * 0.0%) == **0.0%**



Unit testing is the process of testing that a particular function returns expected results for differer of inputs. A unit test is supposed to test standard inputs and corner cases. A unit test should only the logic defined inside the tested function. Most calls to additional functions should be mocked,

especially if they make network or database calls.

(/)

The goal of a unit test is to answer the question: if everything defined outside this function works as expected, does this function work as expected?

Integration tests aim to test a code path end-to-end. In general, only low level functions that make external calls such as HTTP requests, file I/O, database I/O, etc. are mocked.

Integration tests will test interactions between every part of your code.

Execute your tests with

```
$ python -m unittest path/to/test_file.py
```

Resources

Read or watch:

- unittest — Unit testing framework (/rltoken/a_AEObGK8jeqPtTPmm-glA)
- unittest.mock — mock object library (/rltoken/PKetnACd7FfRiU8_kpe5EA)
- How to mock a readonly property with mock? (/rltoken/2ueVPK1kWZuz525FvZ1v2Q)
- parameterized (/rltoken/ml7qc3Y42aZ7GTlLXDxgEg)
- Memoization (/rltoken/x83Hdr54q4Vax5xQ2Z3HSA)

Learning Objectives

At the end of this project, you are expected to be able to explain to anyone (/rltoken/NfT-nNKrNHGrDMY-Qm-1Dg), **without the help of Google**:

- The difference between unit and integration tests.
- Common testing patterns such as mocking, parametrizations and fixtures

Requirements

- All your files will be interpreted/compiled on Ubuntu 18.04 LTS using `python3` (version 3.7)
- All your files should end with a new line
- The first line of all your files should be exactly `#!/usr/bin/env python3`
- A `README.md` file, at the root of the folder of the project, is mandatory
- Your code should use the `pycodestyle` style (version 2.5)
- All your files must be executable
- All your modules should have a documentation (`python3 -c 'print(__import__("my_module").__doc__)'`)
- All your classes should have a documentation (`python3 -c 'print(__import__("my_module").MyClass.__doc__)'`)
- All your functions (inside and outside a class) should have a documentation (`python3 -c 'print(__import__("my_module").my_function.__doc__)'` and `python3 -c 'print(__import__("my_module").MyClass.my_function.__doc__)'`)
- A documentation is not a simple word, it's a real sentence explaining what's the purpose of the module, class or method (the length of it will be verified)
- All your functions and coroutines must be type-annotated.

Required Files

utils.py (or download (https://intranet-projects-files.s3.amazonaws.com/webstack/utils.py))

Click to show/hide file contents

client.py (or download (https://intranet-projects-files.s3.amazonaws.com/webstack/client.py))

Click to show/hide file contents

fixtures.py (or download (https://intranet-projects-files.s3.amazonaws.com/webstack/fixtures.py))

Click to show/hide file contents

Tasks

0. Parameterize a unit test

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Familiarize yourself with the `utils.access_nested_map` function and understand its purpose. Play with it in the Python console to make sure you understand.

In this task you will write the first unit test for `utils.access_nested_map`.

Create a `TestAccessNestedMap` class that inherits from `unittest.TestCase`.

Implement the `TestAccessNestedMap.test_access_nested_map` method to test that the method returns what it is supposed to.

Decorate the method with `@parameterized.expand` to test the function for following inputs:

```
nested_map={"a": 1}, path=("a",)
nested_map={"a": {"b": 2}}, path=("a",)
nested_map={"a": {"b": 2}}, path=("a", "b")
```

For each of these inputs, test with `assertEqual` that the function returns the expected result.

The body of the test method should not be longer than 2 lines.

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`
- File: `test_utils.py`

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

1. Parameterize a unit test

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Implement `TestAccessNestedMap.test_access_nested_map_exception` . Use the `assertRaises` context manager to test that a `KeyError` is raised for the following inputs (use `@parameterized.expand`):

```
nested_map={}, path=("a",)
nested_map={"a": 1}, path=("a", "b")
```

Also make sure that the exception message is as expected.

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`
- File: `test_utils.py`

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

2. Mock HTTP calls

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Familiarize yourself with the `utils.get_json` function.

Define the `TestGetJson(unittest.TestCase)` class and implement the `TestGetJson.test_get_json` method to test that `utils.get_json` returns the expected result.

We don't want to make any actual external HTTP calls. Use `unittest.mock.patch` to patch `requests.get` . Make sure it returns a `Mock` object with a `json` method that returns `test_payload` which you parametrize alongside the `test_url` that you will pass to `get_json` with the following inputs:

```
test_url="http://example.com", test_payload={"payload": True}
test_url="http://holberton.io", test_payload={"payload": False}
```

Test that the mocked `get` method was called exactly once (per input) with `test_url` as argument.

Test that the output of `get_json` is equal to `test_payload` .

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`

- File: **test_utils.py**

(/)

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

3. Parameterize and patch

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Read about memoization and familiarize yourself with the **utils.memoize** decorator.

Implement the **TestMemoize(unittest.TestCase)** class with a **test_memoize** method.

Inside **test_memoize** , define following class

```
class TestClass:

  def a_method(self):
    return 42

  @memoize
  def a_property(self):
    return self.a_method()
```

Use **unittest.mock.patch** to mock **a_method** . Test that when calling **a_property** twice, the correct result is returned but **a_method** is only called once using **assert_called_once** .

Repo:

- GitHub repository: **alx-backend-python**
- Directory: **0x03-Unittests_and_integration_tests**
- File: **test_utils.py**

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

4. Parameterize and patch as decorators

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Familiarize yourself with the **client.GithubOrgClient** class.

In a new **test_client.py** file, declare the **TestGithubOrgClient(unittest.TestCase)** class and implement the **test_org** method.

This method should test that **GithubOrgClient.org** returns the correct value.

Use **@patch** as a decorator to make sure **get_json** is called once with the expected argument but make sure it is not executed.

Use `@parameterized.expand` as a decorator to parametrize the test with a couple of `org` examples to pass to `GithubOrgClient`, in this order:

- google
- abc

Of course, no external HTTP calls should be made.

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`
- File: `test_client.py`

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

5. Mocking a property

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

`memoize` turns methods into properties. Read up on how to mock a property (see resource).

Implement the `test_public_repos_url` method to unit-test `GithubOrgClient._public_repos_url`.

Use `patch` as a context manager to patch `GithubOrgClient.org` and make it return a known payload.

Test that the result of `_public_repos_url` is the expected one based on the mocked payload.

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`
- File: `test_client.py`

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

6. More patching

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Implement `TestGithubOrgClient.test_public_repos` to unit-test `GithubOrgClient.public_repos`.

Use `@patch` as a decorator to mock `get_json` and make it return a payload of your choice.

Use `patch` as a context manager to mock `GithubOrgClient._public_repos_url` and return a value of your choice.

Test that the list of repos is what you expect from the chosen payload.

Test that the mocked property and the mocked `get_json` was called once.

(/)

Repo:

- GitHub repository: **alx-backend-python**
- Directory: **0x03-Unittests_and_integration_tests**
- File: **test_client.py**

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

7. Parameterize

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Implement **TestGithubOrgClient.test_has_license** to unit-test **GithubOrgClient.has_license** .

Parametrize the test with the following inputs

```
repo={"license": {"key": "my_license"}}, license_key="my_license"
repo={"license": {"key": "other_license"}}, license_key="my_license"
```

You should also parameterize the expected returned value.

Repo:

- GitHub repository: **alx-backend-python**
- Directory: **0x03-Unittests_and_integration_tests**
- File: **test_client.py**

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

8. Integration test: fixtures

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

We want to test the **GithubOrgClient.public_repos** method in an integration test. That means that we will only mock code that sends external requests.

Create the **TestIntegrationGithubOrgClient(unittest.TestCase)** class and implement the **setUpClass** and **tearDownClass** which are part of the **unittest.TestCase** API.

Use **@parameterized_class** to decorate the class and parameterize it with fixtures found in **fixtures.py** . The file contains the following fixtures:

```
org_payload, repos_payload, expected_repos, apache2_repos
```

The **setupClass** should mock **requests.get** to return example payloads found in the fixtures.

Use `patch` to start a patcher named `get_patcher`, and use `side_effect` to make sure the mock of `requests.get(url).json()` returns the correct fixtures for the various values of `url` that you anticipate to receive.

Implement the `tearDownClass` class method to stop the patcher.

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`
- File: `test_client.py`

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review

9. Integration tests

#advanced

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `test_public_repos` method to test `GithubOrgClient.public_repos`.

Make sure that the method returns the expected results based on the fixtures.

Implement `test_public_repos_with_license` to test the `public_repos` with the argument `license="apache-2.0"` and make sure the result matches the expected value from the fixtures.

Repo:

- GitHub repository: `alx-backend-python`
- Directory: `0x03-Unittests_and_integration_tests`
- File: `test_client.py`

Done? | Check your code | Ask for a new correction | Get a sandbox | QA Review