SEO Tech Developer

Unit Testing, Integration Testing, and Systems Testing

What you will be able to do:

- Write tests to check if webpages exist
- Write tests that check form submissions
- Create an integration test with a
- Create a mock object to test function behaviors
- Implement BDD Systems Testing using Selenium

SEO Tech Developer

SEO Tech Developer

Unit Testing Webpages

Unit Testing Webpages

- You can unit test a number of aspects of a website
  - Does certain pages/URLs exist?
  - Does a form behave as expected?

- Create a `tests` folder
  - Create `test_basic.py` to test URLs
  - Create `test_users.py` to test the registration form

SEO Tech Developer

Test for Webpage (test_basic.py)

```
import unittest, sys

sys.path.append('../change-to-your-repo-name') # imports python file from parent directory
from main_py_file_name import app #imports flask app object

class BasicTests(unittest.TestCase):

    # executed prior to each test
    def setUp(self):
        self.app = app.test_client()

    def test_main_page(self):
        response = self.app.get('/', follow_redirects=True)
        self.assertEqual(response.status_code, 200)

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

SEO Tech Developer

```
python3 tests/test_basic.py
```

Add Tests for Other Webpages

```
…

def test_main_page(self):
    response = self.app.get('/', follow_redirects=True)
    self.assertEqual(response.status_code, 200)

def test_about_page(self):
    response = self.app.get('/about', follow_redirects=True)
    self.assertEqual(response.status_code, 200)

def test_register_page(self):
    response = self.app.get('/register', follow_redirects=True)
    self.assertEqual(response.status_code, 200)

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

SEO Tech Developer

GitHub Actions (.github/workflows/test.yaml)

```yaml
name: Tests
on: push

jobs:
  unit-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup python
        uses: actions/setup-python@v3
        with:
          python-version: 3.11.3

      - name: Install tools
        run: pip3 install -r requirements.txt

      - name: Test webpages
        run: python3 tests/test_basic.py
```

SEO Tech Developer

Test Form Submission (test_users.py)

```
import unittest, sys, os
sys.path.append('../change-to-your-repo-name') # imports python file from parent directory
from main_py_file_name import app, db #imports flask app object and db app object

class UsersTests(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
        self.app = app.test_client()
        with app.app_context():
          db.drop_all()
          db.create_all()

    def register(self, username, email, password):
        return self.app.post('/register', data=dict(username=username, email=email, password=password, confirm_password=password), follow_redirects=True)

    def test_valid_user_registration(self):
        response = self.register('test', 'test@example.com', 'FlaskIsAwesome')
        self.assertEqual(response.status_code, 200)

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

SEO Tech Developer

```
python3 tests/test_users.py
```

Add Form Tests (test_users.py)

```
def test_valid_user_registration(self):
    response = self.register('test', 'test@example.com', 'FlaskIsAwesome')
    self.assertEqual(response.status_code, 200)

 def test_invalid_username_registration(self):
    response = self.register('t', 'test@example.com', 'FlaskIsAwesome')
    self.assertIn(b'Field must be between 2 and 20 characters long.', response.data)
    response = self.register('thisIsMoreThan20Characters', 'test@example.com', 'FlaskIsAwesome')
    self.assertIn(b'Field must be between 2 and 20 characters long.', response.data)

def test_invalid_email_registration(self):
    response = self.register('test2', 'test@example', 'FlaskIsAwesome')
    self.assertIn(b'Invalid email address.', response.data)
    response = self.register('test3', 'testexample.com', 'FlaskIsAwesome')
    self.assertIn(b'Invalid email address.', response.data)
```

SEO Tech Developer

```
python3 tests/test_users.py
```

Add new tests to GitHub Actions (.github/workflows/test.yaml)

```
- name: Test webpages
  run: python3 tests/test_basic.py

- name: Test registration form
  run: python3 tests/test_users.py
```

SEO Tech Developer

SEO Tech Developer

Integration Testing

Integration Testing

- Integration tests check the interaction between modules.
  - You've actually already written an integration test when you tested whether your website forms worked -- you were testing you main application module, the forms module, the HTML code, and the database connection!
  - For the most part, integration tests are written using the same libraries as unit tests -- the big difference is how much of the code is covered in each test.
- Integration tests are important because...
  - when a developer builds a module, they have a different understanding than the devs building interacting modules
  - interfaces between technology such as modules and a database, are error-prone
  - lack of handling of exceptions can cause interactions to crash in ways that weren't discovered in unit testing
- Approaches to integration testing:
  - **Bottom-up integration testing** starts by testing the modules without dependencies on other modules. This makes fault localization easier which is an advantage, but critical, top-level modules are tested last.
  - **Top-down integration testing** follows the control flow of the system. The critical, top-level modules are first which is an advantage, but it requires a lot of stubbing.

Fakes

- Vocabulary from Week 1:
  - A **fake** can refer to either a mock or a stub - any piece of code that is pretending to be fully implemented, production code.
  - A **mock** is a fake object that mimics an actual object
  - A **stub** can replace an object that isn't built yet

- A stub will never fail a unit test, but a mock can.
- A stub could be replaced when the functionality is added.

SEO Tech Developer

Stubbing

- There are generally 2 times you stub code out:

  1. You need classes and methods to exist for syntactical correctness but they don't need to do anything. You can handle these cases with the `pass` keyword.

     ```
     def future_function(*args):
         pass
     ```

  2. Your yet-to-be-implemented code to do something. In this case, since you know the test cases, you write the minimal code needed to pass the tests. This often looks like a series of conditionals:

     ```
     def fibonacci(n):
       if n == 1:
         return 1
       elif n == 6
         return 8
       else:
         return 233
     ```

SEO Tech Developer

Mocks

- Mocks are useful for controlling the behavior of your code. For example, if you are working on a function that updates someone's Instagram account, when you're testing this function, you don't want to actually change the person's Instagram account. You can create a **mock**, or an imitator, of a person's Instagram account and use the dummy object for altering and testing.
-
- Alternatively, you can create a mock of the update function and use it to update some other piece of that that is not the persons actual Instagram account.

- We can also use mocks to see:
- * When functions were called in our program
- * How many times specific functions were called in our program
- * What arguments were passed to a function when it was called.

SEO Tech Developer

Mock Object Library

- The `unittest.mock` library provides us with:
  - the **`Mock()`** class - creates a fake object which creates methods and attributes when you access them
  - the **`patch()`** methods - gives us the ability to look up an object in a given module and replace it with a Mock object

- You will explore this more in the activity block!

SEO Tech Developer

SEO Tech Developer

Systems Testing

Systems Testing

- Systems testing is when you test the entire system (as opposed to a couple of modules like integration testing)

- Other terms you might hear:
  - **acceptance testing** – when the user tests that the system meets their use case or their business requirements
  - **end-to-end testing** – when you test the system on production or the specific hardware the user will be interacting with

- Systems tests check both front end elements like the web interface and backend like the database

SEO Tech Developer

Behavior Driven Development

- Behavior-driven development (or BDD) is an agile software development technique

- Encourages collaboration between developers, QA and non-technical or business participants in a software project

- BDD focuses on obtaining a clear understanding of desired software behavior

Behave Library

- Behave operates on directories containing:
  - **feature** files written by your Business Analyst with your behavior scenarios

```
Scenario: Search for an account
      Given I search for a valid account
      Then I will see the account details
```

  - a **steps** directory with Python step implementations for the scenarios
    - Example on next slide
  - (optionally) environmental controls (code to run before and after steps, scenarios, features or the whole testing suite)

SEO Tech Developer

Behave Library - Steps

```
@given('I search for a valid account’)
def step_impl(context):
      context.browser.get('http://localhost:8000/index’)
      form = get_element(context.browser, tag='form’)
      get_element(form, name="msisdn").send_keys('61415551234’)
      form.submit()

@then('I will see the account details’)
def step_impl(context):
      elements = find_elements(context.browser, id='no-account’)
      eq_(elements, [], 'account not found’)
      h = get_element(context.browser, id='account-head’)
      ok_(h.text.startswith("Account 61415551234"),
            'Heading %r has wrong text' % h.text)
```

We need a library to interact with our website from python

SEO Tech Developer

```
Scenario: Search for an account
     Given I search for a valid account
     Then I will see the account details
```

Interacting with UI - Selenium

- Selenium is a popular browser automation framework
  - https://github.com/SeleniumHQ/selenium

- There is nice support for Selenium in:
  - C#
  - JavaScript
  - Java
  - Python
  - Ruby

- If you take a look at their docs, Selenium can be quite verbose

SEO Tech Developer

Helium – A Selenium Wrapper

- To make things a little more readable, we can use Helium, a Selenium Wrapper, for common steps
  - Example script:

```
from helium import *

start_chrome("google.com")
write(“seo tech developer")
press(ENTER)
```

SEO Tech Developer

BDD Example with Behave and Helium

- features/environment.py

```
from helium import *

def before_feature(context, feature):
    start_chrome()

def after_feature(context, feature):
    kill_browser()
```

SEO Tech Developer

BDD Example with Behave and Helium

- features/google_search.feature

```
Feature: testing google

  Scenario: visit google and search
     When we visit google
     And search for "Behave"
     Then it should have a title "Behave"
```

SEO Tech Developer

BDD Example with Behave and Helium

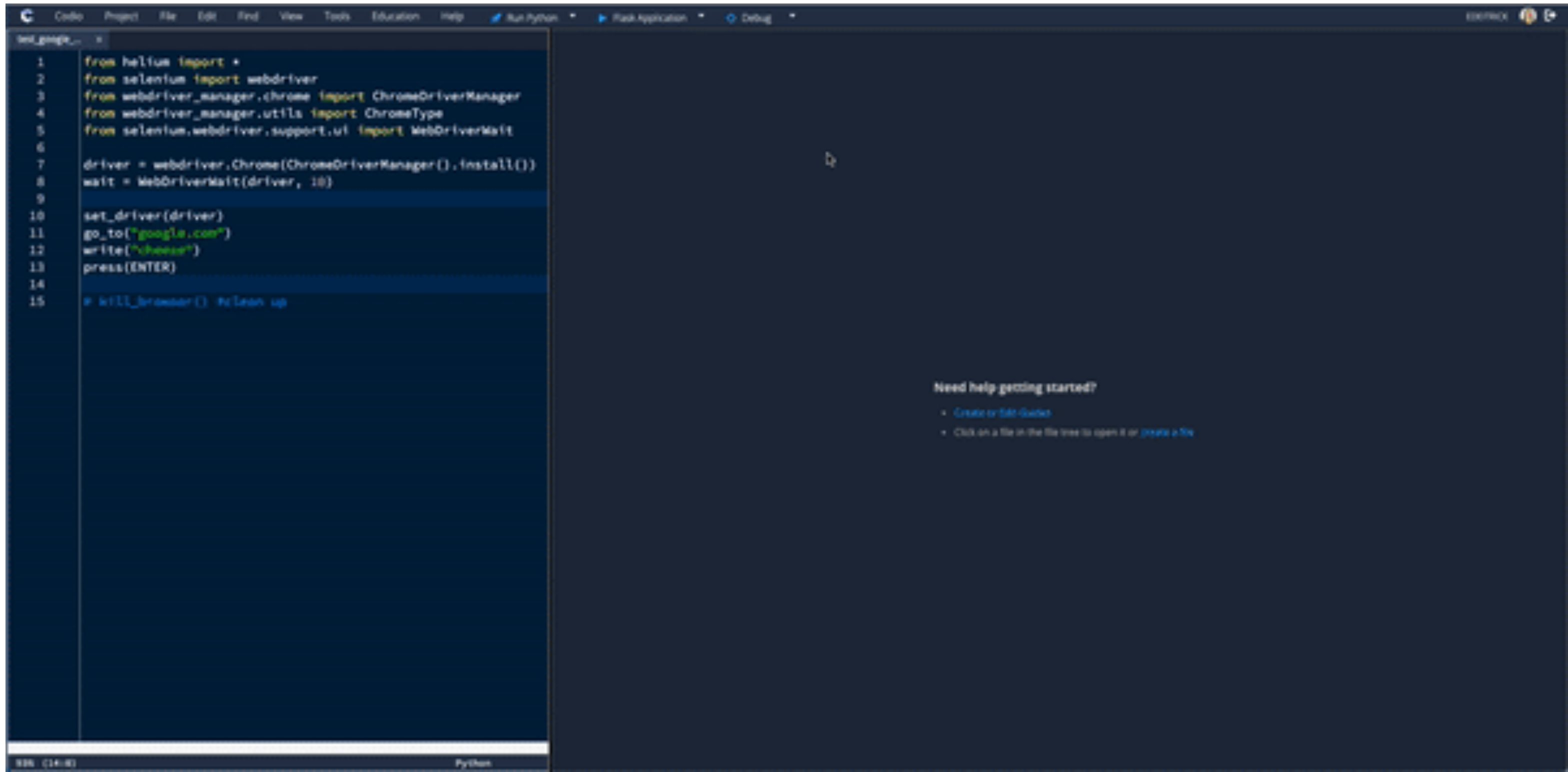
- features/steps/google_steps.py

```python
from helium import *
from behave import then, when
from selenium.webdriver.support.ui import WebDriverWait

@when('we visit google')
def step_impl(context):
    go_to('http://www.google.com')

@when(u'search for "{text}"')
def step_impl(context, text):
    write(text)
    press(ENTER)
    WebDriverWait(Window(), 10)

@then(u'it should have a title "{text}"')
def step_impl(context, text):
    print("" + text + " " + Window().title)
    assert text in Window().title
```

SEO Tech Developer

BDD Example with Behave and Helium

- To run,
  - Open the Virtual Desktop using button on top menu
  - `cd` into `features` and run `behave`
  - You should see a Chrome browser pop up as your tests run
  - Test results are printed out in the browser



```python
from helium import *
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from webdriver_manager.utils import ChromeType
from selenium.webdriver.support.ui import WebDriverWait

driver = webdriver.Chrome(ChromeDriverManager().install())
wait = WebDriverWait(driver, 10)

set_driver(driver)
go_to("google.com")
write("cheese")
press(ENTER)

# kill_browser() #clean up
```

SEO Tech Developer

What questions do you have about…

- Writing tests to check if webpages exist
- Writing tests that check form submissions
- Creating an integration test with a
- Creating a mock object to test function behaviors
- Implementing BDD Systems Testing using Selenium

SEO Tech Developer

SEO Tech Developer

Thank you!