Testing, CI/CD

Testing

- As web applications become increasingly sophisticated and complex, it becomes increasingly important to thoroughly test them. Testing ensures that changes to a function used in many places donâ€™t cause completely different parts of the application to break. Or, functions may behave differently for different types of input, so all types of input should be tested to ensure that the application handles them well.
- Hereâ€™s a basic function, completely separate from any web application, to showcase the idea of testing.

```
import math

def is_prime(n):
    """Determines if a non-negative integer is prime."""
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)):
        if n % i == 0:
            return False
    return True
```

- This function checks if an integer is prime by ensuring that itâ€™s above 2 (the smallest prime) and that it is not evenly divisble by any number less than its square root.
- One way to test this function would be to simply run this function in the Python interpreter and manually test it. While this might work for a small example, this will ultimately become tedious. The next best step is to write a test function.

```
from prime import is_prime

def test_prime(n, expected):
  if is_prime(n) != expected:
      print(f"ERROR on is_prime({n}), expected {expected}")
```

- A simple way to use this test function is to write a short Python program that runs this test for a series of inputs. That might look like this:

```
from prime import is_prime

def test_prime(n, expected):
    if is_prime(n) != expected:
        print(f"ERROR on is_prime({n}), expected {expected}")

if __name__ == "__main__":
    test_prime(-4, False)
    test_prime(-3, False)
    test_prime(-2, False)
    test_prime(-1, False)
    test_prime(0, False)
    test_prime(1, False)
    test_prime(2, True)
    test_prime(3, True)
    test_prime(4, False)
```

- This can be run in the terminal with `python tests0.py`.
- Now that the testing is automated, when a change is made to `prime.py`, it is easy to see if the problem is solved.

```
for i in range(2, int(math.sqrt(n) + 1):
    if n % i == 0:
        return False
```

- This was a simple off-by-1 error. `range` returns values starting at the first argument and up to, but not including, the second.
- Running the test file again after this change will produce no errors. This does not mean the function is totally correct, only that the given tests have been passed. Now, the challenge is to write good, comprehensive tests that cover all

possible conditions. In this case, that might mean testing even and odd numbers, etc. This becomes increasingly important for larger applications.
- Remember also that we do not necessarily only run tests when we think things might break. Sometimes, we might do so after *refactoring* our code, trying to optimize its functionality. The above `is_prime()` function could, perhaps, be optimized in a more â€œPythonicâ€ way (Python is somewhat notorious for examples like this) as follows:

```
import math

def is_prime(n):
  return n > 1 and all(n % i for i in range(2, int(math.sqrt(n)) + 1))
```

- Breaking this down, there are two conditions being checked:
  - Is `n` greater than 1; and
  - Is it the case that all of the values from 2 up to (and including) the square root have a â€œtruthyâ€ value of `True`? (If so, that means `n % i` is nonzero for all of them; i.e., `n` is not evenly divisible by any of them.)
- If, and only if, both of those are true, is the number considered prime!

assert

- A useful Python feature for testing is the built-in `assert` command, which is followed by an boolean expression. If it does not evaluate to `True`, Python will throw an `AssertionError`.
- All programs, on exit, return an exit code. Generally speaking, an exit code of 0 indicates that everything went well, and any other code indicates an error. To examine an exit code in bash after running a Python script, use `echo $?`.

unittest

- `unittest` is a built-in Python library for testing. Testing `is_prime` with `unittest` might look like this:

```
import unittest

from prime import is_prime


class Tests(unittest.TestCase):

    def test_1(self):
        """Check that 1 is not prime."""
        self.assertFalse(is_prime(1))

    def test_2(self):
        """Check that 2 is prime."""
        self.assertTrue(is_prime(2))

    def test_8(self):
        """Check that 8 is not prime."""
        self.assertFalse(is_prime(8))

    def test_11(self):
        """Check that 11 is prime."""
        self.assertTrue(is_prime(11))

    def test_25(self):
        """Check that 25 is not prime."""
        self.assertFalse(is_prime(25))

    def test_28(self):
        """Check that 28 is not prime."""
        self.assertFalse(is_prime(28))


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

  - `Tests` inherits from `unittest.TestCase`, which signifies that it contains a series of tests, each of which is capable of extending the basic functionality defined in `unittest.TestCase`.
  - Each test inside `Tests` is simply a method with an appropriate â€œdocstringâ€ labelling it.

    - `unittest` has a series of built-in, more advanced and readable assert statements. Instead of using `assert isPrime(1) == False`, simply use `self.assertFalse(is_prime(1))`.
    - `unittest.main()` will run all the tests.
- `unittest` methods include (but are not limited to):
    - `assertEqual`
    - `assertNotEqual`
    - `assertTrue`
    - `assertFalse`
    - `assertIn` : checks if an item is in a list
    - `assertNotIn` : checks if an item is not in a list

Testing Web Applications with Django

The Back-End

- Django has its own testing framework to make it easy to test web applications. Test code is found in the application directory in `tests.py`, the one file we did not really consider in Lecture 7.
- Hereâ€™s a function that could be in the `Flight` model and that might need to be tested if it is going to be used in a view.

```
def is_valid_flight(self):
    return (self.origin != self.destination) and (self.duration >= 0)
```

    - This returns `True` if the origin and the destination arenâ€™t the same and its duration is positive.
- Hereâ€™s an example `flights/tests.py`:

```
from django.test import TestCase

from .models import Airport, Flight

# Create your tests here.
class ModelsTestCase(TestCase):

    def setUp(self):

        # Create airports.
        a1 = Airport.objects.create(code="AAA", city="City A")
        a2 = Airport.objects.create(code="BBB", city="City B")

        # Create flights.
        Flight.objects.create(origin=a1, destination=a2, duration=100)
        Flight.objects.create(origin=a1, destination=a1, duration=200)
        Flight.objects.create(origin=a1, destination=a2, duration=-100)

    def test_departures_count(self):
        a = Airport.objects.get(code="AAA")
        self.assertEqual(a.departures.count(), 3)

    def test_arrivals_count(self):
        a = Airport.objects.get(code="AAA")
        self.assertEqual(a.arrivals.count(), 1)

    def test_valid_flight(self):
        a1 = Airport.objects.get(code="AAA")
        a2 = Airport.objects.get(code="BBB")
        f = Flight.objects.get(origin=a1, destination=a2, duration=100)
        self.assertTrue(f.is_valid_flight())

    def test_invalid_flight_destination(self):
        a1 = Airport.objects.get(code="AAA")
        f = Flight.objects.get(origin=a1, destination=a1)
        self.assertFalse(f.is_valid_flight())

    def test_invalid_flight_duration(self):
        a1 = Airport.objects.get(code="AAA")
        a2 = Airport.objects.get(code="BBB")
        f = Flight.objects.get(origin=a1, destination=a2, duration=-100)
        self.assertFalse(f.is_valid_flight())
```

    - `TestCase` is a extension to the `unittest` framework that makes it easier to test some Django application specific things.
    - `ModelsTestCase` is a class that, like before, contains functions for every test.
    - In the `TestCase` framework, the `setUp` function will be run before any tests. In this case, some airports and flights are created for the tests to check.
        - The `setUp` function is actually run before every test, to ensure that the tests are independent.
- Using Djangoâ€™s infrastructure to run this test is quite powerful because it makes it easy to run all tests across all applications and, because Django knows tests are likely to involve databse manipulations, it creates a separate test database that tests can interact with. This means that `setUp` functions like the one above wonâ€™t mess up the real database by trying to add fake airports and flights for testing purposes.
- To run the tests, run simply `python manage.py test`.

Defending Against Bad Data

- We can also write methods in our models to prevent against illogical or â€œbadâ€ data. One instance, for example, might be to try to prevent content managers from trying to create flights with the same origin and destination, or a non-positive duration. To do this, we can override the functionality of some more built-in Django testing functions.
- In `airline1/models.py`:

```
# Add a method that raises "Validation errors" if the data is illogical.
def clean(self):
    if self.origin == self.destination:
        raise ValidationError("Origin and destination must be different.")
    elif self.duration < 1:
        raise ValidationError("Duration must be positive.")

# Call this method before trying to add data, overriding the default behavior of built-in `save`.
def save(self, *args, **kwargs):
    self.clean()

    # This syntax now calls Django's own "save" function, adding this data to the DB (if `clean` was ok).
    super().save(*args, **kwargs)
```

The Front End

- Now that the models have been tested, the next step is to test the views.

```
from django.db.models import Max
from django.test import Client, TestCase

from .models import Airport, Flight, Passenger

# Create your tests here.
class FlightsTestCase(TestCase):

    # ...same setUp and model testing as before...

    def test_index(self):
        c = Client()
        response = c.get("/")
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.context["flights"].count(), 2)

    def test_valid_flight_page(self):
        a1 = Airport.objects.get(code="AAA")
        f = Flight.objects.get(origin=a1, destination=a1)

        c = Client()
        response = c.get(f"/{f.id}")
        self.assertEqual(response.status_code, 200)

    def test_invalid_flight_page(self):
        max_id = Flight.objects.all().aggregate(Max("id"))["id__max"]

        c = Client()
        response = c.get(f"/{max_id + 1}")
        self.assertEqual(response.status_code, 404)
```

```
            def test_flight_page_passengers(self):
                f = Flight.objects.get(pk=1)
                p = Passenger.objects.create(first="Alice", last="Adams")
                f.passengers.add(p)

                c = Client()
                response = c.get(f"/{f.id}")
                self.assertEqual(response.status_code, 200)
                self.assertEqual(response.context["passengers"].count(), 1)

            def test_flight_page_non_passengers(self):
                f = Flight.objects.get(pk=1)
                p = Passenger.objects.create(first="Alice", last="Adams")

                c = Client()
                response = c.get(f"/{f.id}")
                self.assertEqual(response.status_code, 200)
                self.assertEqual(response.context["non_passengers"].count(), 1)
```

- `Client` simulates a web client that, for testing purposes, can make requests to and get responses from a web server. Using `Client`, requests to different pages can be simulated to ensure that the expected information is being returned.
- `c.get("/")` simply uses a `Client` object to make a `GET` request to a route and returns the response (stored as `response`). This response can be checked by verifying `response.status_code` and the contents of `response.contexts`.
- An argument can be passed to a URL using the same curly brace/dot notation syntax as before.
- `Flight.objects.all().aggregate(Max("id"))["id__max"]` returns the maximum ID value of any flight. This is for test the response to an invalid flight ID in a URL.

Selenium

- For testing browser behavior, including JavaScript code, a separate browser testing tool is necessary. One such tool is Selenium, which uses a web driver that allows Python code to programatically pretend to be a user interacting with a webpage. Hereâ€™s an example webpage to test that has a counter that can be incremented or decremented with two buttons:

```
<html>
    <head>
        <title>Counter</title>
        <script>

            document.addEventListener('DOMContentLoaded', () =>  {

                let counter = 0;

                document.querySelector('#increase').onclick = () => {
                    counter++;
                    document.querySelector('h1').innerHTML = counter;
                };

                document.querySelector('#decrease').onclick = () => {
                    counter--;
                    document.querySelector('h1').innerHTML = counter;
                };
            });
        </script>
    </head>
    <body>
        <h1>0</h1>
        <button id="increase">+</button>
        <button id="decrease">-</button>
    </body>
</html>
```

- Hereâ€™s the Selenium Python code to test the page:

```
import os
import pathlib
import unittest
```

```
from selenium import webdriver

# A convenience function to turn a filename into a full path, as needed for a browser
def file_uri(filename):
    return pathlib.Path(os.path.abspath(filename)).as_uri()

driver = webdriver.Chrome()

class WebpageTests(unittest.TestCase):

    def test_title(self):
        driver.get(file_uri("counter.html"))
        self.assertEqual(driver.title, "Counter")

    def test_increase(self):
        driver.get(file_uri("counter.html"))
        increase = driver.find_element_by_id("increase")
        increase.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "1")

    def test_decrease(self):
        driver.get(file_uri("counter.html"))
        decrease = driver.find_element_by_id("decrease")
        decrease.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "-1")

    def test_multiple_increase(self):
        driver.get(file_uri("counter.html"))
        increase = driver.find_element_by_id("increase")
        for i in range(3):
            increase.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "3")


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

- `file_uri` takes in an HTML file and returns the URL that would access that file.
- `webdriver.Chrome()` is one of many built-in Selenium web drivers. This one in particular is for interacting with Google Chrome.
- `driver.get` will open up whatever URL is passed in.
- Each button is programmatically tested by finding each `button` element by ID and calling the `click` function to simulate a user clicking on it. Then, whatever the text display is can get verified to match the expected value.

Continuous Integration and Continuous Delivery

- CI (continuous integration) consists of frequent integration and merging of code changes between different project contributors back to a main branch along with automated unit testing to verify these integrations. Likewise, CD (continuous delivery) consists of making frequent, incremental updates to a web application as those updates are finished.
- There are many different tools with the purpose of facilitating CI and testing. One of the more popular ones and the one used in this class is Travis. When code is pushed to GitHub, GitHub will notify Travis of those changes. Travis will pull that code and run some tests on it. GitHub will then be notified of the test results.
- Travisâ€™s configuration file, which lists any tests, installations, etc., is written in the YAML file format. YAML files are composed of keys and values, similar to JSON.

```
key1: value1
key2: value2
key3:
    - item1
    - item2
    - item3
key4:
    nested_key1: value3
    nested_key2:
        - item4
        - item5
```

- In particular, a very simple Travis YAML file will look something like this:

```
language: python
python: 3.6
install: pip install -r requirements.txt
script: python manage.py test
```

- `install` lists the commands that should be run to install any necessary components before testing. Listing any requirements, such as Django, in `requirements.txt` will automate that installation.
- `script` lists the command for actually running the tests.
- To actually configure Travis, go to https://travis-ci.org and sync a GitHub account. Then, any repositories that should be tracked by Travis can be selected. After making a push to GitHub, it will be visible on the Travis website as a â€˜buildâ€™ and will execute the commands as dictated in the configuration file. Travis is able to check whether or not tests were passed by checking the exit code of the testing command. If a build fails any tests, this will be marked on GitHubâ€™s commit log with a red X. A build currently being tested will be marked with a yellow dot, and a successful build will be marked with a green check.

Continuous Deployment

Deploying our app to Heroku:

- To creating a Heroku app, create an account if you donâ€™t have one or login if you do.
- Create a new app specifying its name which has to be unique and its region.
- Generating a Heroku API key to Authorize Travis CI:
  - Head to **Applications** tab under the account settings.
  - Under **Authorizations**, click Create Authorization.
  - Add a description and optionally specify a duration (in seconds) after which this API key will expire (you likely donâ€™t want to expire this if you intend for your project to run indefinitely) then click Create.
- Provisioning a PostgreSQL database:
  - Head to your appâ€™s resources dashboard which should have a URL of the form `https://dashboard.heroku.com/apps/<APP-NAME>/resources` where APP NAME is the actual name of your app. In our example, it was `https://dashboard.heroku.com/apps/kzidane-airline/resources` for example.
- Under **Add-ons** find Heroku Postgres.
- Select the item and click Provision to create your PostgreSQL database.
- Finding the database credentials:
  - Click on the Database that you just created then click the Settings tab near the top.
  - Next to Database Credentials click **View Credentials**.
- Setting environment variables *for the Heroku app*:
  - In your app settings which should be at `https://dashboard.heroku.com/apps/<APP NAME>/settings`, click **Reveal Config Vars**.
  - Add the following variables (recall these are accessed in airline4/airline/settings.py in the demo):
    - `DATABASE_USER` (should be the value of User from database credentials)
    - `DATABASE_PASSWORD` (should be the value of Password from database credentials)
    - `DATABASE_NAME` (should be the value of Database from database credentials)
    - `DATABASE_HOST` (should be the value of Host from database credentials)
    - `DATABASE_PORT` (should be the value of Port from database credentials)
- Viewing the logs (to see project behavior)
  - In your app dashboard which should be at `https://dashboard.heroku.com/apps/<APP NAME>` click on More on the top-right then click View logs.

Required files for Heroku

- At the root of your Django project folder, you have to have a `requirements.txt` listing any Python package dependencies your project uses (e.g., Django itself), one per line.
- You also must have a file called `Procfile`, which looks like the below:

```
web: gunicorn <PROJECT NAME>.wsgi
```

- where `PROJECT NAME` is the actual project name (for example, `web: gunicorn airline.wsgi`) to let Heroku know how to serve your project.

Using Travis to allow for Continuous Delivery of our Application

- Adding environment variables to your project:
- While logged into Travis CI, head to `https://travis-ci.com/USERNAME/REPOSITORY/settings` where `USERNAME` is your actual GitHub username and `REPOSITORY` is the name of your repository. In our example, it was `https://travis-ci.com/web50student1/airline4/settings`.
- In the *Environment Variables* section add the following environment variables:
    - `DATABASE_USER` whose value is `postgres`
    - `DATABASE_PASSWORD` whose value is `postgres`
    - `DATABASE_NAME` whose value is `testdb`
    - `DATABASE_HOST` whose value is `0.0.0.0`
    - `DATABASE_PORT` whose value is `5432`
    - `HEROKU_API_KEY` whose value is the Heroku API key you generated above.
- The first five of the above needed to be added on Travis to allow it to perform **tests**, but these are not our production credentials; thatâ€™s why those were added to Heroku, before!
- The sixth of these is an environment variable that Travis needs in order to actually deploy our code once it finishes testing (without it, anyone could deploy to our Heroku app â€“ probably not ideal!)
- Lastly, we need to teach Travis to deploy our code after testing it. To do so, we need to modify our `.travis.yml` file somewhat. At the end of your .travis.yml add the following keys and values to have Travis CI deploy to your Heroku app after a successful build of the master branch:

```
deploy:
  provider: heroku
  api_key: $HEROKU_API_KEY
  app: APP
  run: python manage.py migrate
  on: master
```

Where `APP` is the name of your actual Heroku app as you specified above.