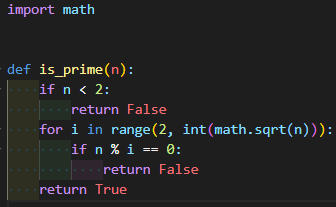
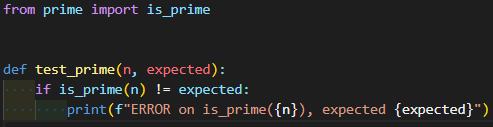
**<Lecture7. Testing, CI/CD>**

**Test-Driven Development**

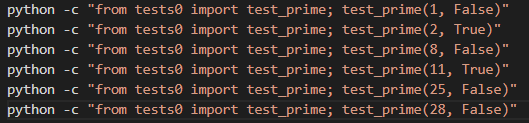
**# prime.py**



**# tests0.py**

****

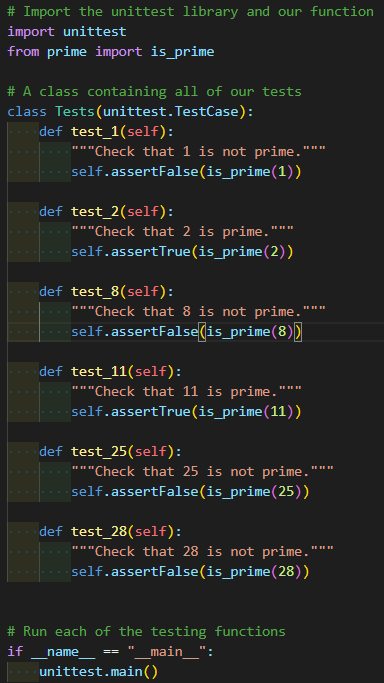
**# tests0.sh**

****

./tests0.sh 로 실행

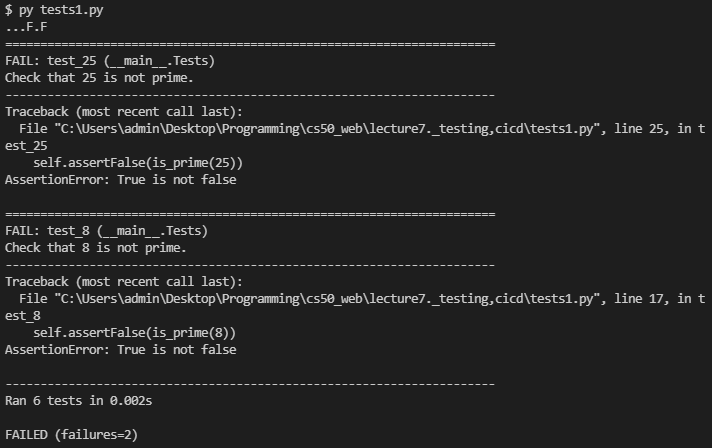


**Unit Testing**

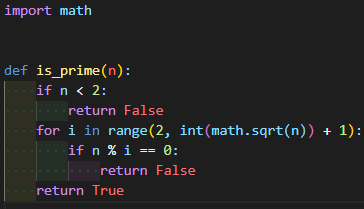


Notice that each of the functions within our Tests class followed a pattern:

* The name of the functions begin with test\_. This is necessary for the functions to be run automatically with the call to unittest.main().
* Each test takes in the self argument. This is standard when writing methods within Python classes.
* The first line of each function contains a docstring surrounded by three quotation marks. These are not just for the code’s readability. When the tests are run, the comment will be displayed as a discription of the test if it fails.
* The next line of each of the functions contained an assertion in the form self.assertSOMETHING. There are many different assertions you can make including assertTrue, assertFalse, assertEqual, and assertGreater. You can find these ones and more by checking out the documentation. Notice that each of the functions within our Tests class followed a pattern:



**# prime.py**

****

range 범위를 수정하여 error를 fix한다.

**Django Testing & Client Testing**

We’re first going to add a method to our Flight model that verifies that a flight is valid by checking for two conditions:

1. The origin is not the same as the destination
2. The duration is greater than 0 minutes

we will probably want to check not just whether or not specific functions work, but also whether or not individual web pages load as intended. We can do this by creating a Client object in our Django testing class, and then making requests using that object. To do this, we’ll first have to add Client to our imports:

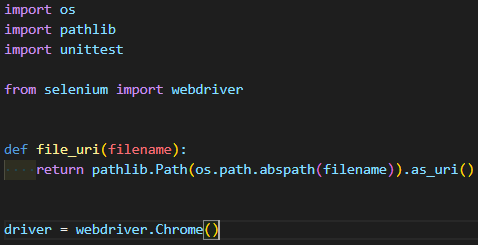
For example, let’s now add a test that makes sure that we get an HTTP response code of 200 and that all three of our flights are added to the context of a response:



**Selenium**

Now if we wish to test this code, we could just open up our web browser, click the two buttons, and observe what happens. This, however, would become very tedious as you write larger and larger single page applications, which is why several frameworks have been created that help with in-browser testing, one of which is called Selenium.

Using Selenium, we’ll be able to define a testing file in Python where we can simulate a user opening a web browser, navigating to our page, and interacting with it. Our main tool when doing this is known as a Web Driver, which will open up a web browser on your computer. Let’s take a look at how we could start using this library to begin interacting with pages. Note that below we use both selenium and ChromeDriver. Selenium can be installed for python by running pip install selenium, and ChromeDriver can be installed by running pip install chromedriver-py



The above code is all of the basic setup we need, so now we can get into some more interesting uses by employing the Python interpreter. One note about the first few lines is that in order to target a specific page, we need that page’s Uniform Resource Identifier (URI) which is a unique string that represents that resource.

>>> from tests import \*

# Find the URI of our newly created file

>>> uri = file\_uri("counter.html")

# Use the URI to open the web page

>>> driver.get(uri)

# Access the title of the current page

>>> driver.title

'Counter'

# Access the source code of the page

>>> driver.page\_source

'<html lang="en"><head>\n <title>Counter</title>\n <script>\n \n // Wait for page to load\n document.addEventListener(\'DOMContentLoaded\', () => {\n\n // Initialize variable to 0\n let counter = 0;\n\n // If increase button clicked, increase counter and change inner html\n document.querySelector(\'#increase\').onclick = () => {\n counter ++;\n document.querySelector(\'h1\').innerHTML = counter;\n }\n\n // If decrease button clicked, decrease counter and change inner html\n document.querySelector(\'#decrease\').onclick = () => {\n counter --;\n document.querySelector(\'h1\').innerHTML = counter;\n }\n })\n </script>\n </head>\n <body>\n <h1>0</h1>\n <button id="increase">+</button>\n <button id="decrease">-</button>\n \n</body></html>'

# Find and store the increase and decrease buttons:

>>> increase = driver.find\_element\_by\_id("increase")

>>> decrease = driver.find\_element\_by\_id("decrease")

# Simulate the user clicking on the two buttons

>>> increase.click()

>>> increase.click()

>>> decrease.click()

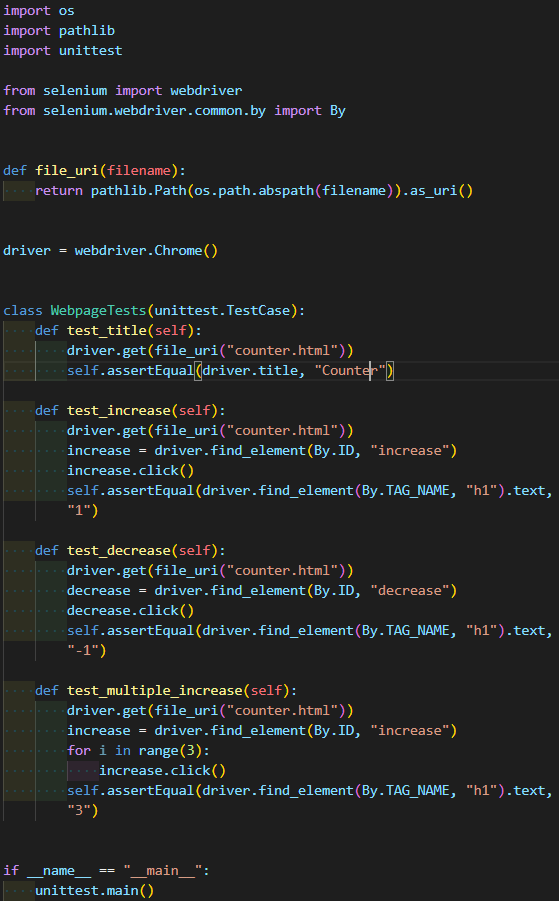
# We can even include clicks within other Python constructs:

>>> for i in range(25):

... increase.click()

일일이 클릭하지 않아도 콘솔창에 명령문을 입력하여 기능을 test할 수 있다.

**# selenium/tests.py**

****

**unittest Methods**

assertEqual/NotEqual

assertTrue/False

assertIn/NotIn

…

**CI/CD**

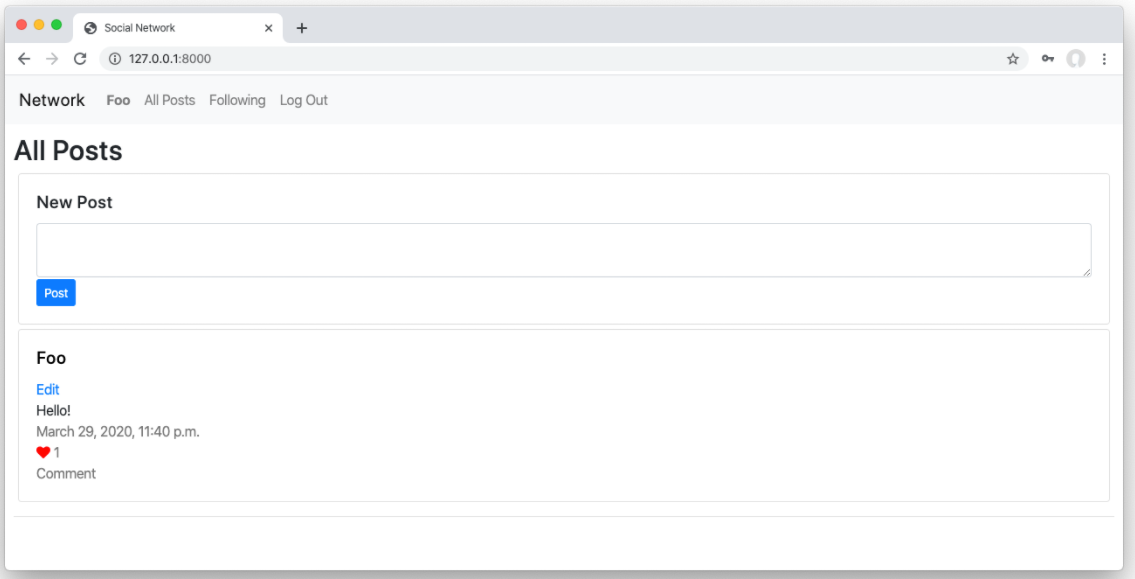
CI/CD, which stands for Continuous Integration and Continuous Delivery, is a set of software development best practices that dictate how code is written by a team of people, and how that code is later delivered to users of the application. As the name implies, this method consists of two main parts:

* Continuous Integration:
  + Frequent merges to the main branch
  + Automated unit testing with each merge
* Continuous Delivery:
  + Short release schedules, meaning new versions of an application are released frequently.

CI/CD has become more and more popular among software development teams for a number of reasons:

* When different team members are working on different features, many compatibility issues can arise when multiple features are combined at the same time. Continuous integration allows teams to tackle small conflicts as they come.
* Because unit tests are run with each Merge, when a test fails it is easier to isolate the part of the code that is causing the problem.
* Frequently releasing new versions of an application allows developers to isolate problems if they arise after launch.
* Releasing small, incremental changes allows users to slowly get used to new app features rather than being overwhelmed with an entirely different version
* Not waiting to release new features allows companies to stay ahead in a competitive market.

**<Project 4. Network>**

****

**Specification**

Using Python, JavaScript, HTML, and CSS, complete the implementation of a social network that allows users to make posts, follow other users, and “like” posts. You must fulfill the following requirements:

* New Post: Users who are signed in should be able to write a new text-based post by filling in text into a text area and then clicking a button to submit the post.
  + The screenshot at the top of this specification shows the “New Post” box at the top of the “All Posts” page. You may choose to do this as well, or you may make the “New Post” feature a separate page.
* All Posts: The “All Posts” link in the navigation bar should take the user to a page where they can see all posts from all users, with the most recent posts first.
  + Each post should include the username of the poster, the post content itself, the date and time at which the post was made, and the number of “likes” the post has (this will be 0 for all posts until you implement the ability to “like” a post later).
* Profile Page: Clicking on a username should load that user’s profile page. This page should:
  + Display the number of followers the user has, as well as the number of people that the user follows.
  + Display all of the posts for that user, in reverse chronological order.
  + For any other user who is signed in, this page should also display a “Follow” or “Unfollow” button that will let the current user toggle whether or not they are following this user’s posts. Note that this only applies to any “other” user: a user should not be able to follow themselves.
* Following: The “Following” link in the navigation bar should take the user to a page where they see all posts made by users that the current user follows.
  + This page should behave just as the “All Posts” page does, just with a more limited set of posts.
  + This page should only be available to users who are signed in.
* Pagination: On any page that displays posts, posts should only be displayed 10 on a page. If there are more than ten posts, a “Next” button should appear to take the user to the next page of posts (which should be older than the current page of posts). If not on the first page, a “Previous” button should appear to take the user to the previous page of posts as well.
  + See the Hints section for some suggestions on how to implement this.
* Edit Post: Users should be able to click an “Edit” button or link on any of their own posts to edit that post.
  + When a user clicks “Edit” for one of their own posts, the content of their post should be replaced with a textarea where the user can edit the content of their post.
  + The user should then be able to “Save” the edited post. Using JavaScript, you should be able to achieve this without requiring a reload of the entire page.
  + For security, ensure that your application is designed such that it is not possible for a user, via any route, to edit another user’s posts.
* “Like” and “Unlike”: Users should be able to click a button or link on any post to toggle whether or not they “like” that post.
  + Using JavaScript, you should asynchronously let the server know to update the like count (as via a call to fetch) and then update the post’s like count displayed on the page, without requiring a reload of the entire page.

**Hints**

* For examples of JavaScript fetch calls, you may find some of the routes in Project 3 useful to reference.
* You’ll likely need to create one or more models in network/models.py and/or modify the existing User model to store the necessary data for your web application.
* Django’s Paginator class may be helpful for implementing pagination on the back-end (in your Python code).
* Bootstrap’s Pagination features may be helpful for displaying pages on the front-end (in your HTML).