

CS50 使用 Python 和 JavaScript 进行 Web 编程

开放式课程

捐  (<https://cs50.harvard.edu/donate>)


Brian Yu (<https://brianyu.me>)

brian@cs.harvard.edu

大卫·J·马兰 (<https://cs.harvard.edu/malan/>)

malan@harvard.edu

 (<https://www.facebook.com/dmalan>)  (<https://github.com/dmalan>) 

(<https://www.instagram.com/davidjmalan/>)  (<https://www.linkedin.com/in/malan/>)

 (<https://www.reddit.com/user/davidjmalan>) 

(<https://www.threads.net/@davidjmalan>)  (<https://twitter.com/davidjmalan>)

讲座 7

- [介绍](#)
- [测试](#)
- [断言](#)
 - [测试驱动开发](#)
- [单元测试](#)
- [Django 测试](#)
 - [客户端测试](#)
- [硒](#)
- [持续集成/持续交付](#)
- [GitHub 操作](#)
- [Docker](#)

介绍

- 到目前为止，我们已经讨论了如何使用 HTML 和 CSS 构建简单的网页，以及如何使用 Git 和 GitHub 来跟踪代码更改并与他人协作。我们还熟悉了 Python 编程语言，开始使用 Django 创建 Web 应用程序，并学习了如何使用 Django 模型在我们的网站中存储信

息。然后，我们介绍了 JavaScript，并学习了如何使用它使网页更具交互性，并讨论了使用动画和 React 进一步改进我们的用户界面。

- 今天，我们将学习开展和启动大型项目的最佳实践。

测试

软件开发过程的一个重要部分是**测试**我们编写的代码，以确保一切按预期运行。在本讲座中，我们将讨论几种可以改进代码测试方式的方法。

断言

在 Python 中运行测试的最简单方法之一是使用命令 `assert`。此命令后跟一些应该是的表达式 `True`。如果表达式是 `True`，则不会发生任何事情，如果是 `False`，则会引发异常。让我们看看如何结合命令来测试 `square` 我们在第一次学习 Python 时编写的函数。当函数正确编写时，不会发生任何事情，因为 `assert` 是 `True`

```
def square(x):  
    return x * x  
  
assert square(10) == 100  
  
""" Output:  
"""
```

然后当写错误时，就会引发异常。

```
def square(x):  
    return x + x  
  
assert square(10) == 100  
  
""" Output:  
Traceback (most recent call last):  
  File "assert.py", line 4, in <module>  
    assert square(10) == 100  
AssertionError  
"""
```

测试驱动开发

当您开始构建大型项目时，您可能需要考虑使用**测试驱动开发**，这是一种开发风格，每次修复错误时，您都会将检查该错误的测试添加到不断增长的测试集中，这些测试会在每次进行更改时运行。这将帮助您确保添加到项目中的附加功能不会干扰现有功能。

现在，让我们看一个稍微复杂一点的函数，思考一下编写测试如何帮助我们发现错误。我们现在编写一个名为 `is_prime` 的函数，当且仅当其输入为素数时，它才 `is_prime` 返回： `True`

```
import math

def is_prime(n):

    # We know numbers less than 2 are not prime
    if n < 2:
        return False

    # Checking factors up to sqrt(n)
    for i in range(2, int(math.sqrt(n))):

        # If i is a factor, return false
        if n % i == 0:
            return False

    # If no factors were found, return true
    return True
```

现在，让我们看一下我们编写的用于测试我们 `prime` 功能的函数：

```
from prime import is_prime

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

此时，我们可以进入我们的 Python 解释器并测试一些值：

```
>>> test_prime(5, True)
>>> test_prime(10, False)
>>> test_prime(25, False)
ERROR on is_prime(25), expected False
```

从上面的输出中我们可以看出，5 和 10 被正确识别为素数和非素数，但 25 被错误地识别为素数，所以我们的函数一定有问题。不过，在研究函数的问题之前，让我们先看看自动化测试的方法。我们可以做到这一点的一种方法是创建一个 **shell 脚本**，或者一些可以在终端内运行的脚本。这些文件需要 `.sh` 扩展名，所以我们的文件将被称为 `tests0.sh`。下面每一行都包含以下内容

1. A `python3` 指定我们正在运行的 Python 版本
2. A `-c` 表示我们希望运行命令
3. 以字符串格式运行的命令

```
python3 -c "from tests0 import test_prime; test_prime(1, False)"
python3 -c "from tests0 import test_prime; test_prime(2, True)"
python3 -c "from tests0 import test_prime; test_prime(8, False)"
python3 -c "from tests0 import test_prime; test_prime(11, True)"
python3 -c "from tests0 import test_prime; test_prime(25, False)"
python3 -c "from tests0 import test_prime; test_prime(28, False)"
```

`./tests0.sh` 现在我们可以终端中运行这些命令，得到以下结果：

```
ERROR on is_prime(8), expected False
ERROR on is_prime(25), expected False
```

单元测试

尽管我们能够使用上述方法自动运行测试，但我们仍然可能希望避免编写每个测试。幸运的是，我们可以使用 Python `unittest` 库使这个过程变得更容易一些。让我们看看我们的 `is_prime` 函数的测试程序可能是什么样子。

```
# Import the unittest library and our function
import unittest
from prime import is_prime

# A class containing all of our tests
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))

# Run each of the testing functions
if __name__ == "__main__":
    unittest.main()
```

请注意，我们类中的每个函数都 `Tests` 遵循一种模式：

- 函数名称以 开头 `test_`。这是通过调用 来自动运行函数所必需的 `unittest.main()`。
- 每个测试都接受参数 `self`。这是在 Python 类中编写方法时的标准。
- 每个函数的第一行包含一个由三个引号包围的**文档字符串**。这不仅仅是为了提高代码的可读性。运行测试时，如果测试失败，则注释将显示为测试的描述。

- 每个函数的下一行都包含一个断言，格式为 `self.assertSOMETHING`。您可以做出许多不同的断言，包括 `assertTrue`、`assertFalse`、`assertEqual` 和 `assertGreater`。您可以通过查看[文档](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertEqual) (<https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertEqual>)找到这些以及更多内容。

现在，让我们看看这些测试的结果：

```
...F.F
=====
FAIL: test_25 (__main__.Tests)
Check that 25 is not prime.
-----
Traceback (most recent call last):
  File "tests1.py", line 26, in test_25
    self.assertFalse(is_prime(25))
AssertionError: True is not false

=====
FAIL: test_8 (__main__.Tests)
Check that 8 is not prime.
-----
Traceback (most recent call last):
  File "tests1.py", line 18, in test_8
    self.assertFalse(is_prime(8))
AssertionError: True is not false

-----
Ran 6 tests in 0.001s

FAILED (failures=2)
```

运行测试后，它会向我们提供一些有关测试发现的有用信息。在第一行中，它会按照测试编写的顺序，`unittest` 为我们提供一系列 `.` 表示成功和失败的 `s`。 `F`

```
...F.F
```

接下来，对于每个失败的测试，我们都会得到失败的函数的名称：

```
FAIL: test_25 (__main__.Tests)
```

我们之前提供的描述性评论：

```
Check that 25 is not prime.
```

以及异常的回溯：

```
Traceback (most recent call last):
  File "tests1.py", line 26, in test_25
    self.assertFalse(is_prime(25))
AssertionError: True is not false
```

最后，我们将了解进行了多少次测试、花费了多少时间以及有多少次测试失败：

```
Ran 6 tests in 0.001s

FAILED (failures=2)
```

现在让我们看看如何修复函数中的错误。事实证明，我们需要在循环中测试一个额外的数字 `for`。例如，当时 `n`，25 平方根是 5，但是当它是函数中的一个参数时 `range`，`for` 循环在数字处终止 4。因此，我们可以简单地将循环的标题更改 `for` 为：

```
for i in range(2, int(math.sqrt(n)) + 1):
```

现在，当我们使用单元测试再次运行测试时，我们得到以下输出，表明我们的更改修复了该错误。

```
.....
-----
Ran 6 tests in 0.000s

OK
```

随着您努力优化此函数，这些自动化测试将变得更加有用。例如，您可能希望利用这样一个事实：您不需要检查所有整数作为因数，只需检查较小的素数（如果某个数字不能被 3 整除，那么它也不能被 6、9、12 等整除），或者您可能希望使用更高级的概率素数测试，例如 Fermat [和 \(https://en.wikipedia.org/wiki/Fermat_primality_test\)](https://en.wikipedia.org/wiki/Fermat_primality_test) Miller-Rabin [素数测试 \(https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test\)](https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test)。每当您进行更改以改进此函数时，您都希望能够轻松地再次运行单元测试，以确保您的函数仍然正确。

Django 测试

现在，让我们看看在创建 Django 应用程序时如何应用自动化测试的思想。在处理此问题时，我们将使用 `flights` 首次了解 Django 模型时创建的项目。我们首先要向模型添加一种方法 `Flight`，通过检查两个条件来验证航班是否有效：

1. 出发地与目的地不同
2. 时长大于0分钟

现在，我们的模型看起来像这样：

```
class Flight(models.Model):
    origin = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="departures")
    destination = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="arrivals")
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origin} to {self.destination}"
```

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

为了确保我们的应用程序按预期工作，每当我们创建一个新的应用程序时，都会自动为我们提供一个 `tests.py` 文件。当我们第一次打开此文件时，我们会看到 Django 的 `TestCase` (<https://docs.djangoproject.com/en/4.0/topics/testing/overview/>) 库已自动导入：

```
from django.test import TestCase
```

使用该 `TestCase` 库的一个优点是，当我们运行测试时，将创建一个全新的数据库，仅用于测试目的。这很有用，因为我们避免了意外修改或删除数据库中现有条目的风险，并且我们不必担心删除仅为测试而创建的虚拟条目。

要开始使用这个库，我们首先要导入所有模型：

```
from .models import Flight, Airport, Passenger
```

然后我们将创建一个新类来扩展 `TestCase` 我们刚刚导入的类。在这个类中，我们将定义一个 `setUp` 将在测试过程开始时运行的函数。在这个函数中，我们可能想要创建。我们的类开始时的样子如下：

```
class FlightTestCase(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)
```

现在我们的测试数据库中有一些条目，让我们向此类添加一些函数来执行一些测试。首先，让我们通过尝试计算从机场出发的航班数量（我们知道应该是 3）和到达的航班数量（应该是 1）来确保 `departures` 和 `arrivals` 字段正常工作：

```
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)
```

我们还可以测试 `is_valid_flight` 添加到 `Flight` 模型中的函数。我们首先断言当航班有效时该函数确实返回 `true`：

```
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())
```

接下来，让我们确保目的地和持续时间无效的航班返回 false：

```
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())
```

现在，为了运行测试，我们将运行 `python manage.py test`。此输出与我们在使用 Python 库时看到的输出几乎相同 `unittest`，尽管它还记录了它正在创建和销毁测试数据库：

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..FF.
=====
FAIL: test_invalid_flight_destination (flights.tests.FlightTestCase)
-----
Traceback (most recent call last):
  File "/Users/cleggett/Documents/cs50/web_notes_files/7/django/airline/flights/test
    self.assertFalse(f.is_valid_flight())
AssertionError: True is not false

=====
FAIL: test_invalid_flight_duration (flights.tests.FlightTestCase)
-----
Traceback (most recent call last):
  File "/Users/cleggett/Documents/cs50/web_notes_files/7/django/airline/flights/test
    self.assertFalse(f.is_valid_flight())
AssertionError: True is not false

-----
Ran 5 tests in 0.018s

FAILED (failures=2)
Destroying test database for alias 'default'...
```

从上面的输出中我们可以看出，有时应该 `is_valid_flight` 返回 `True` 但结果却返回了 `False`。进一步检查我们的函数后，我们可以看到，我们犯了一个错误，使用了 `or` 而不是 `and`，这意味着只需满足其中一个航班要求，航班即可有效。如果我们将函数更改为：

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


然后我们可以再次运行测试并获得更好的结果：

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 5 tests in 0.014s

OK
Destroying test database for alias 'default'...
```

客户端测试

在创建 Web 应用程序时，我们可能不仅要检查特定功能是否正常工作，还要检查各个网页是否按预期加载。我们可以通过 `Client` 在 Django 测试类中创建一个对象，然后使用该对象发出请求来实现这一点。为此，我们首先必须添加 `Client` 到我们的导入中：

```
from django.test import Client, TestCase
```

例如，现在让我们添加一个测试，确保我们获得 HTTP 响应代码 200，并且所有三趟航班都添加到响应的上下文中：

```
def test_index(self):

    # Set up client to make requests
    c = Client()

    # Send get request to index page and store response
    response = c.get("/flights/")

    # Make sure status code is 200
    self.assertEqual(response.status_code, 200)

    # Make sure three flights are returned in the context
    self.assertEqual(response.context["flights"].count(), 3)
```

我们可以类似地检查以确保我们获得有效航班页面的有效响应代码，以及不存在的航班页面的无效响应代码。（请注意，我们使用函数 `Max` 来查找最大值 `id`，我们可以通过 `from django.db.models import Max` 在文件顶部包含来访问它）

```
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"/flights/{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"/flights/{max_id + 1}")
self.assertEqual(response.status_code, 404)
```

最后，让我们添加一些测试以确保乘客和非乘客名单按预期生成：

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

现在，我们可以一起运行所有测试，并看到目前没有错误！

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 10 tests in 0.048s

OK
Destroying test database for alias 'default'...
```

硒

到目前为止，我们已经能够测试使用 Python 和 Django 编写的服务器端代码，但在构建应用程序时，我们也希望能够为客户端代码创建测试。例如，让我们回想一下我们的 `counter.html` 页面并为其编写一些测试。

我们首先编写一个略有不同的计数器页面，其中包含一个用于减少计数的按钮：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Counter</title>
    <script>

      // Wait for page to load
      document.addEventListener('DOMContentLoaded', () => {
```

```

// Initialize variable to 0
let counter = 0;

// If increase button clicked, increase counter and change inner html
document.querySelector('#increase').onclick = () => {
    counter++;
    document.querySelector('h1').innerHTML = counter;
}

// If decrease button clicked, decrease counter and change inner html
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>

```

现在，如果我们想测试此代码，我们可以打开 Web 浏览器，单击两个按钮，然后观察会发生什么。但是，随着您编写越来越大的单页应用程序，这将变得非常繁琐，这就是为什么创建了几个有助于浏览器内测试的框架的原因，其中一个称为 [Selenium](https://www.selenium.dev/) (<https://www.selenium.dev/>)。

使用 Selenium，我们将能够在 Python 中定义一个测试文件，在其中模拟用户打开 Web 浏览器、导航到我们的页面并与之交互。执行此操作时，我们的主要工具称为 **Web 驱动程序**，它将在您的计算机上打开 Web 浏览器。让我们看看如何开始使用这个库来开始与页面交互。请注意，下面我们同时使用了 `selenium` 和 `ChromeDriver`。可以通过运行 `pip install selenium` 安装 Selenium，也 `ChromeDriver` 可以通过运行 `pip install chromedriver-py` 安装。

```

import os
import pathlib
import unittest

from selenium import webdriver

# Finds the Uniform Resource Identifier of a file
def file_uri(filename):
    return pathlib.Path(os.path.abspath(filename)).as_uri()

# Sets up web driver using Google chrome
driver = webdriver.Chrome()

```

上面的代码就是我们所需的所有基本设置，现在我们可以使用 Python 解释器来实现一些更有趣的用途。关于前几行的一点需要注意的是，为了定位特定页面，我们需要该页面的**统一资源标识符 (URI)**，它是代表该资源的唯一字符串。

```
# 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

# 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()
```

现在让我们看看如何使用这个模拟来创建页面的自动化测试:

```
# Standard outline of testing class
class WebpageTests(unittest.TestCase):

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

    def test_increase(self):
        """Make sure header updated to 1 after 1 click of increase button"""
        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")

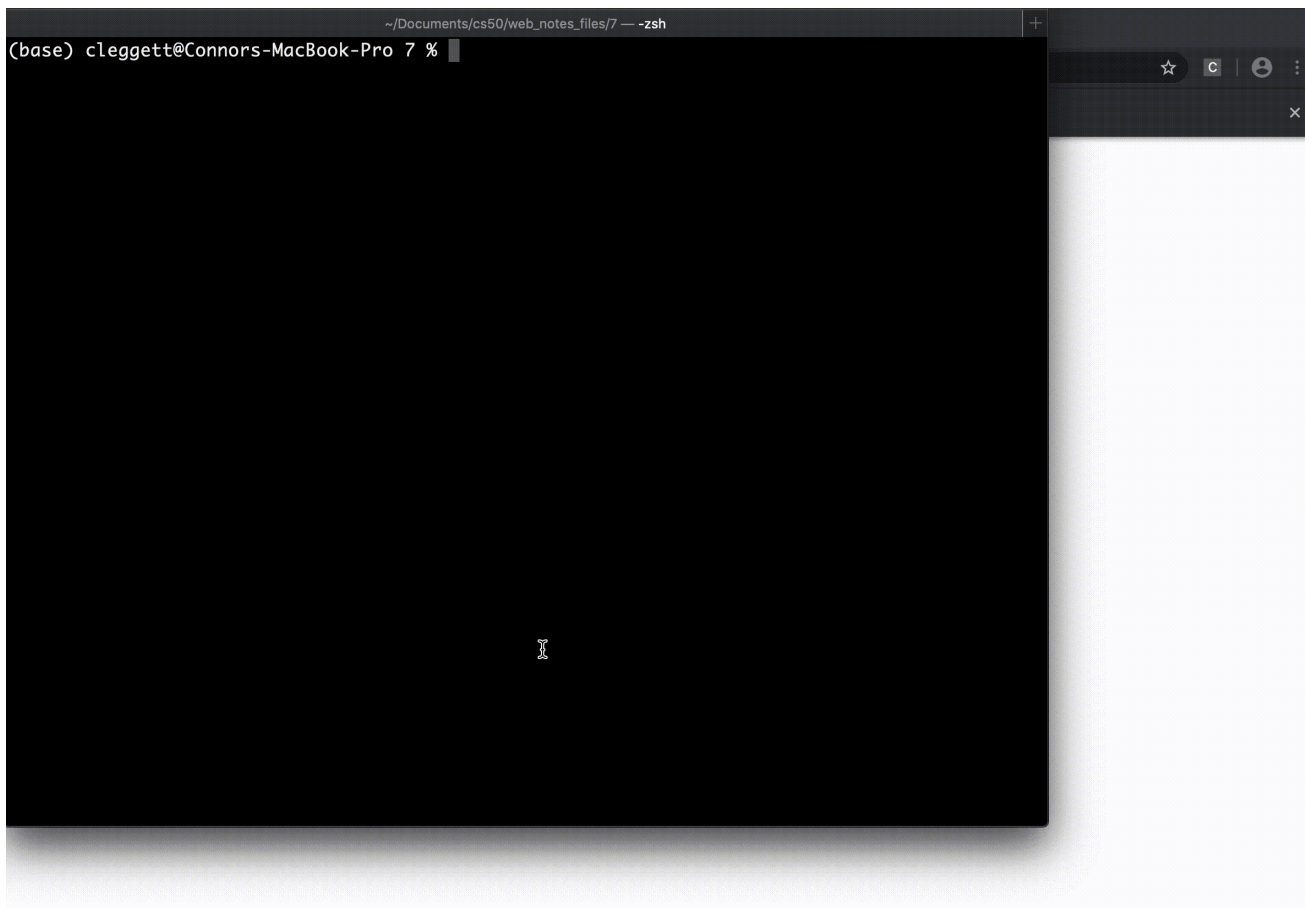
    def test_decrease(self):
        """Make sure header updated to -1 after 1 click of decrease button"""
        driver.get(file_uri("counter.html"))
        decrease = driver.find_element_by_id("decrease")
        decrease.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "-1")

    def test_multiple_increase(self):
        """Make sure header updated to 3 after 3 clicks of increase button"""
        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()
```

现在，如果我们运行 `python tests.py`，我们的模拟将在浏览器中进行，然后测试结果将打印到控制台。以下是代码中存在错误且测试失败时的示例：



持续集成/持续交付

CI/CD代表**持续集成和持续交付**，是一组软件开发最佳实践，规定了团队如何编写代码，以及随后如何将该代码交付给应用程序的用户。顾名思义，此方法由两个主要部分组成：

- 持续集成：
 - 频繁合并到主分支
 - 每次合并时自动进行单元测试
- 持续交付：
 - 发布时间表短，意味着应用程序的新版本会频繁发布。

CI/CD 在软件开发团队中越来越受欢迎，原因如下：

- 当不同的团队成员开发不同的功能时，同时组合多个功能可能会出现许多兼容性问题。持续集成使团队能够随时解决小冲突。

- 由于每次合并都会运行单元测试，因此当测试失败时，更容易隔离导致问题的代码部分。
- 频繁发布应用程序的新版本可以让开发人员隔离发布后出现的问题。
- 发布小的、渐进式的更改可以让用户慢慢习惯新的应用功能，而不是被一个完全不同的版本所淹没
- 不等待发布新功能可以让公司在竞争激烈的市场中保持领先地位。

GitHub 操作

一种用于帮助持续集成的流行工具是 [GitHub Actions](https://github.com/features/actions)。 [GitHub](https://github.com/features/actions) (<https://github.com/features/actions>) Actions 允许我们创建工作流，我们可以在其中指定每次有人推送到 git 存储库时要执行的某些操作。例如，我们可能希望在每次推送时检查是否遵守了样式指南，或者是否通过了一组单元测试。

为了设置 GitHub 操作，我们将使用一种名为 **YAML** 的配置语言。YAML 围绕键值对（如 JSON 对象或 Python 字典）构造其数据。以下是一个简单的 YAML 文件的示例：

```
key1: value1
key2: value2
key3:
  - item1
  - item2
  - item3
```

`name.yml` 现在，让我们看一个示例，了解如何配置与 GitHub Actions 配合使用的 YAML 文件（采用或形式 `name.yml`）。为此，我将 `.github` 在存储库中创建一个目录，然后 `workflows` 在该目录中创建一个目录，最后 `ci.yml` 在该目录中创建一个文件。在该文件中，我们将写入：

```
name: Testing
on: push

jobs:
  test_project:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Django unit tests
        run: |
          pip3 install --user django
          python3 manage.py test
```

由于这是我们第一次使用 GitHub Actions，让我们了解一下该文件的每个部分的作用：

- 首先，我们给出工作流程 `name`，在我们的例子中是测试。
- 接下来，使用 `on` 密钥指定工作流程的运行时间。在我们的例子中，我们希望每次有人推送到存储库时都执行测试。

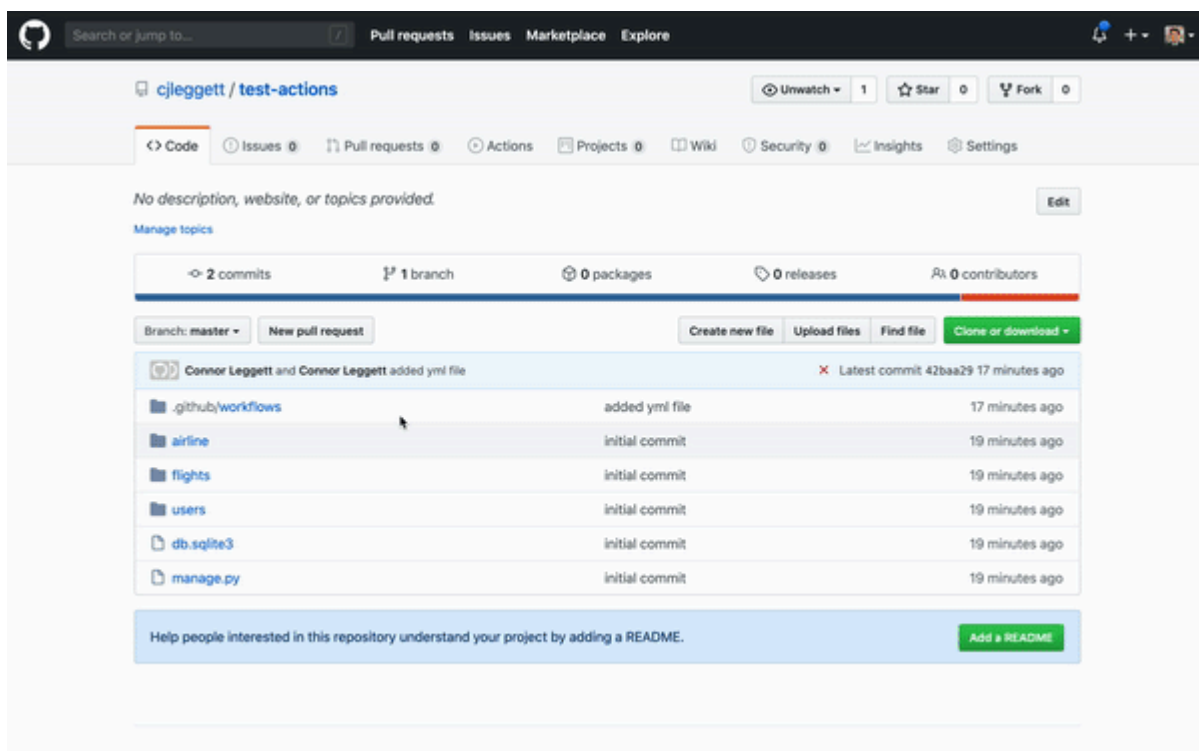
- 文件的其余部分包含在一个 `jobs` 键中，该键指示每次推送时应该运行哪些作业。
 - 在我们的例子中，唯一的工作是 `test_project`。每个工作必须定义两个组件
 - 该 `runs-on` 密钥指定我们希望我们的代码在 GitHub 的哪个虚拟机上运行。
 - 该 `steps` 键提供运行此作业时应该发生的操作
 - 在 `uses` 密钥中我们指定我们希望使用哪个 GitHub 操作。
`actions/checkout@v2` 是我们使用的由 GitHub 编写的操作。
 - 这里的关键 `name` 是让我们提供我们正在采取的行动描述
 - 输入密钥后 `run`，我们输入要在 GitHub 服务器上运行的命令。在我们的例子中，我们希望安装 Django，然后运行测试文件。

现在，让我们打开 GitHub 中的存储库并查看页面顶部附近的一些选项卡：

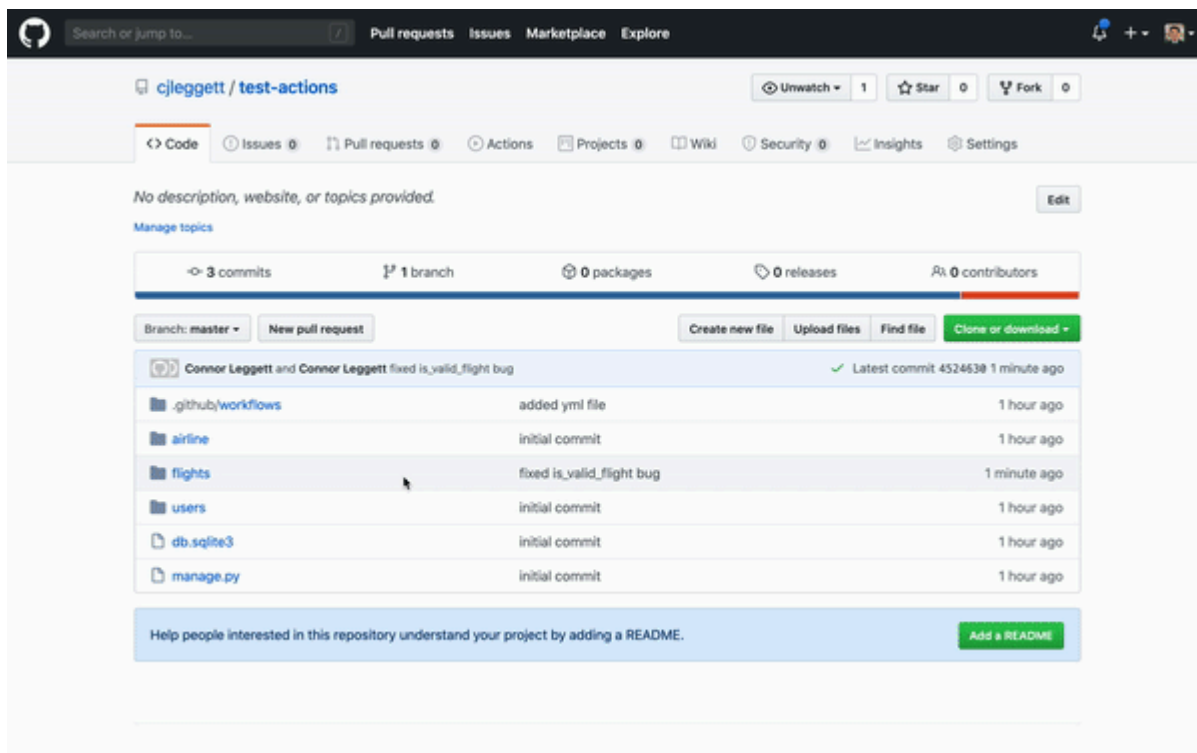
- **代码**：这是我们最常用的选项卡，因为它允许我们查看目录中的文件和文件夹。
- **问题**：在这里我们可以打开和关闭问题，这些问题是修复错误或添加新功能的请求。我们可以将其视为应用程序的待办事项列表。
- **拉取请求**：希望将某个分支中的代码合并到另一个分支的人提出的请求。这是一个非常有用的工具，因为它允许人们在将代码集成到主分支之前进行**代码审查**，发表评论并提供建议。
- **GitHub Actions**：这是我们在进行持续集成时使用的选项卡，因为它提供了每次推送后发生的操作的日志。

在这里，让我们想象一下，在修复项目中函数中的错误之前，我们推送了更改。我们现在可以导航到**GitHub Actions**选项卡，单击我们最近的推送，单击失败的操作，然后查看日志：

`is_valid_flight` `models.py` `airport`



现在，修复了错误之后，我们可以再次运行并找到更好的结果：



Docker

在软件开发领域，当你的计算机配置与运行应用程序的计算机配置不同时，就会出现这个问题。你可能安装了不同版本的 Python 或一些额外的软件包，这些软件包允许应用程序在你的计算机上顺利运行，但在你的服务器上却会崩溃。为了避免这些问题，我们需要一种方法来确保每个参与项目的人都使用相同的环境。一种方法是使用一个名为 **Docker** 的工具，它是一种容器化软件，这意味着它会在你的计算机中创建一个隔离的环境，可以在许多合作者和运行你网站的服务器之间实现标准化。虽然 Docker 有点像 **虚拟机**，但它们实际上是不同的技术。虚拟机（比如在 GitHub Actions 上使用的虚拟机或启动 **AWS** 服务器时使用的虚拟机）实际上是一台拥有自己操作系统的整个虚拟计算机，这意味着它最终会占用大量空间，无论它在哪里运行。另一方面，Docker 通过在现有计算机中设置容器来工作，因此占用的空间更少。

现在我们已经了解了什么是 Docker 容器，让我们看看如何在计算机上配置它。第一步是创建一个 **Docker 文件**，我们将它命名为 `Dockerfile`。在这个文件中，我们将提供如何创建 **Docker 镜像** 的说明，该镜像描述了我们希望包含在容器中的库和二进制文件。下面是我们的镜像 `Dockerfile` 可能的样子：

```
FROM python:3
COPY . /usr/src/app
WORKDIR /usr/src/app
RUN pip install -r requirements.txt
CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]
```

在这里，我们将深入了解上述文件的实际作用：

- `FROM python3`：这表明我们基于安装了 Python 3 的标准镜像来创建此镜像。这在编写 Docker 文件时相当常见，因为它可以让你避免为每个新镜像重新定义相同的基本设置。

- `COPY . /usr/src/app`：这表明我们希望从当前目录（`.`）复制所有内容并将其存储 `/usr/src/app` 在新容器的目录中。
- `WORKDIR /usr/src/app`：这将设置我们在容器内运行命令的位置。（有点像 `cd` 在终端上）
- `RUN pip install -r requirements.txt`：在此行中，假设您已将所有要求都包含在名为的文件中 `requirements.txt`，它们都将安装在容器内。
- `CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]`：最后，我们指定启动容器时应该运行的命令。

到目前为止，在本课程中，我们仅使用了 SQLite，因为这是 Django 的默认数据库管理系统。但是在有真实用户的实时应用程序中，SQLite 几乎从未使用过，因为它不像其他系统那样容易扩展。值得庆幸的是，如果我们希望为数据库运行单独的服务器，我们可以简单地添加另一个 Docker 容器，然后使用名为 **Docker Compose** 的功能将它们一起运行。这将允许两个不同的服务器在单独的容器中运行，但也可以相互通信。为了指定这一点，我们将使用一个名为的 YAML 文件 `docker-compose.yml`：

```
version: '3'

services:
  db:
    image: postgres

  web:
    build: .
    volumes:
      - ../usr/src/app
    ports:
      - "8000:8000"
```

在上面的文件中我们：

- 指定我们使用 Docker Compose 版本 3
- 概述两项服务：
 - `db` 根据 Postgres 已经写入的图像设置我们的数据库容器。
 - `web` 通过指示 Docker 来设置我们服务器的容器：
 - 使用当前目录中的 Dockerfile。
 - 使用容器内指定的路径。
 - 将容器内的8000端口链接到我们电脑上的8000端口。

现在，我们准备使用命令启动我们的服务 `docker-compose up`。这将在新的 Docker 容器内启动我们的两个服务器。

此时，我们可能想要在 Docker 容器中运行命令来添加数据库条目或运行测试。为此，我们首先运行 `docker ps` 以显示所有正在运行的 docker 容器。然后，找到 `CONTAINER ID` 我们想要进入的容器并运行 `docker exec -it CONTAINER_ID bash -l`。这会将您移动到我们在

`usr/src/app` 容器内设置的目录内。我们可以在该容器内运行任何我们想要的命令，然后通过运行退出 `CTRL-D`。

本次讲座就到这里！下次，我们将致力于扩大我们的项目规模并确保其安全。