วิศวกรรมซอฟต์แวร์

Software Engineering

สมเกียรติ วังศิริพิทักษ์

somkiat.wa@kmitl.ac.th

ห้อง 518 หรือ ห้อง 506 (MIV Lab)

PART II

Software Testing & TDD

Test-Driven Development

Test-Driven Development

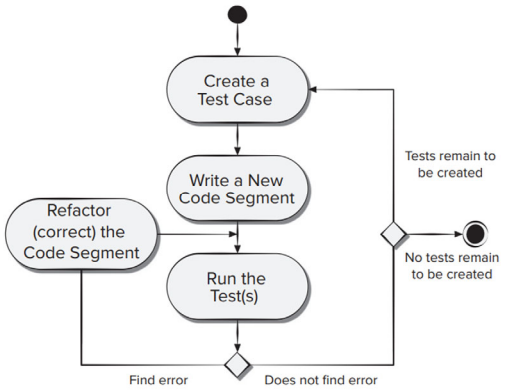
- **Requirements** drive **design**. (Software)
  - **Design** establishes a foundation for construction.
- What about component-level design and construction ?
- In test-driven development (**TDD**) ...
  - Requirements for a software component serve as **the basis for** the creation of **a series of test cases** that ...
    - exercise the interface and
    - attempt to find errors in the data structures and functionality delivered by the component.

สร้างเทสเคสก่อน ก่อนจะโค้ดจริง จาก requirements

Test-Driven Development

- TDD emphasizes the design of **test cases before** the creation of **source code.**
  - TDD process flow
- Before the first small segment of code is created ...

  a software engineer creates a test to exercise the code (to try to make the code fail).
- The code is then written to satisfy the test.

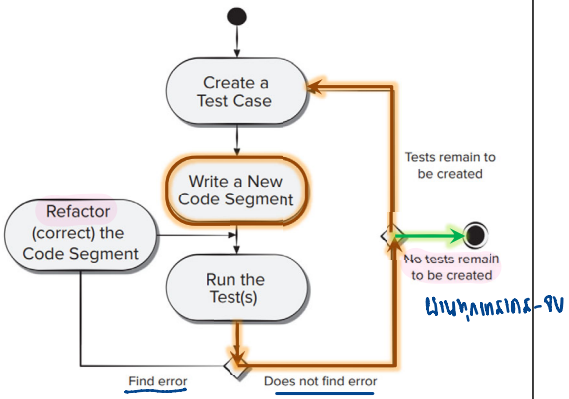


Test-Driven Development

- The test(s) is run.
- If it passes ...

  a new test is created for the next segment of code to be developed.
- The process continues ...

  until the component is fully coded and all tests execute without error.

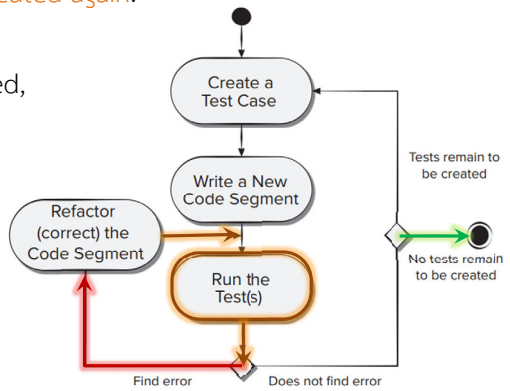


Test-Driven Development

- However, if any test succeeds in finding an error ...

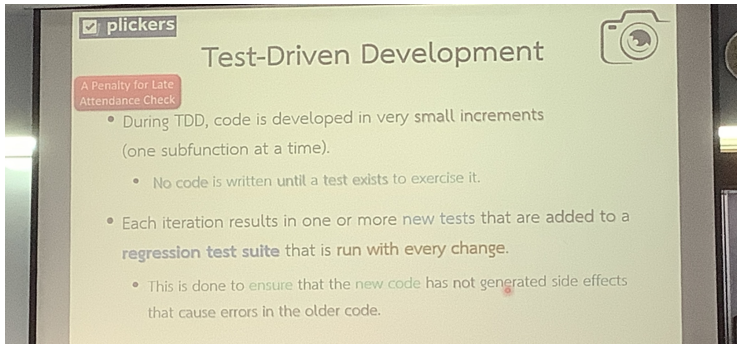
  the existing code is refactored (corrected) and ...

  all tests created to that point are executed again.
- This iterative flow continues until there are no tests left to be created, implying that the component meets all requirements defined for it.

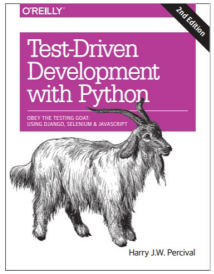


Test-Driven Development

- During TDD, code is developed in very small increments (one subfunction at a time).
  - No code is written until a test exists to exercise it.



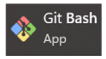
Test-Driven Development with Python



- Take you through the development of a real web application from beginning to end.
  - Learn how to write and run **tests** before building each part of the app, and then ...
  - Develop the minimum amount of code required to pass those tests.
- In the process, you will learn the basics of Django, Selenium, Git, ~~jQuery, and Mock,~~ along with current web development techniques.

Test-Driven Development with Python

- Start 'git-bash.exe'.



- Run

```
w_som@seiYoga7i14ITL5 MINGW64 ~
$ workon superlists
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 ~
$
```

⇔ deactivate

- Download 'geckodriver.exe'
- Put it in the folder '/c/WebDrivers/'
- Run
  echo 'PATH=/c/WebDrivers:$PATH' >> ~/.bashrc
  source ~/.bashrc

virtualenvwrapper

mkvirtualenv -p python3 superlists

pip install django selenium
OR
pip install django --upgrade
pip install selenium --upgrade

Test-Driven Development

The Basics of TDD and Django

The Basics of TDD and Django

- We'll build a real web application from scratch, writing **tests first** at every stage.
  - Note that we will cover only the basics to give you an understanding of the concept of TDD.
- We'll cover **functional testing** with Selenium, as well as **unit testing**, and see the difference between the two.
- Introduce the TDD workflow (unit-test/code cycle).
- We'll also do some refactoring, and see how that fits with TDD.
- We'll also be using a version control system (Git). We'll discuss how and when to do commits and integrate them with the TDD and web development workflow.

Getting Django Set Up Using a Functional Test

- In TDD the first step is **always** the same: **write a test**.
  - First, we write the test.

  - Then we run it and check that it fails as expected.

  - Only then do we go ahead and build some of our app.

- Another thing is to take one step at a time.

Getting Django Set Up Using a Functional Test

- The first thing is **check** that we've got **Django** installed, and that it's ready for us to work with.
  - The way we'll check is by confirming that we can spin up Django's development server and actually see it serving up a web page, in our web browser, on our local PC.
- We'll use the **Selenium** browser automation tool for this.

Test-Driven Development with Python

functional_tests_firefox.py

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get("http://localhost:8000")

assert "The install worked successfully! Congratulations!" in
browser.title
```

Starting a Selenium "webdriver" to pop up a real Firefox browser window

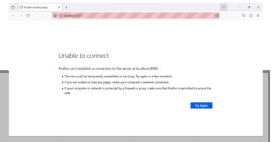
Using it to open up a web page

Checking (making a test assertion) that the page has the word "Django" in its title

- That's our first functional test (FT)
- Let's try running it:

Test-Driven Development with Python

functional_tests_firefox.py



```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture
$ python functional_tests_firefox.py
Traceback (most recent call last):
  File "D:\somkiat\src\SE\Lecture\functional_tests_firefox.py", line 5, in <module>
    browser.get("http://localhost:8000")
  File "C:\Users\w_som\.virtualenvs\superlists\lib\site-
packages\selenium\webdriver\remote\webdriver.py", line 356, in get
    self.execute(Command.GET, {"url": url})
  File "C:\Users\w_som\.virtualenvs\superlists\lib\site-
packages\selenium\webdriver\remote\webdriver.py", line 347, in execute
    self.error_handler.check_response(response)
  File "C:\Users\w_som\.virtualenvs\superlists\lib\site-
packages\selenium\webdriver\remote\errorhandler.py", line 229, in check_response
    raise exception_class(message, screen, stacktrace)
selenium.common.exceptions.WebDriverException: Message: Reached error page:
about:neterror?e=connectionFailure&u=http%3A//localhost%3A8000/&c=UTF-
8&d=Firefox%20can%E2%80%99t%20establish%20a%20connection%20to%20the%20server%20at%20localh
ost%3A8000.
Stacktrace:
RemoteError@chrome://remote/content/shared/RemoteError.sys.mjs:8:8
WebDriverError@chrome://remote/content/shared/webdriver/Errors.sys.mjs:191:5
UnknownError@chrome://remote/content/shared/webdriver/Errors.sys.mjs:800:5
```

A *failing* test

Getting Django Up and Running

- The **first step in getting Django up** and running is to create a project, which will be the main container for our site.

```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture
$ django-admin.exe startproject superlists
```

- That will create a folder called **superlists**, and a set of files and subfolders inside it:
  - The superlists/superlists folder is for stuff that applies to the whole project—like settings.py, for example, which is used to store global configuration information for the site.

```
.
├── funtional_tests_firefox.py
├── geckodriver.log
└── superlists
    ├── manage.py
    └── superlists
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```

Getting Django Up and Running

- manage.py is used to run a development server, etc.
- Let's try that now.

```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture
$ cd superlists

(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture/superlists
$
```

- We'll work from this folder a lot.
- Then run:

```
.
├── funtional_tests_firefox.py
├── geckodriver.log
└── superlists
    ├── manage.py
    └── superlists
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```


Getting Django Up and Running

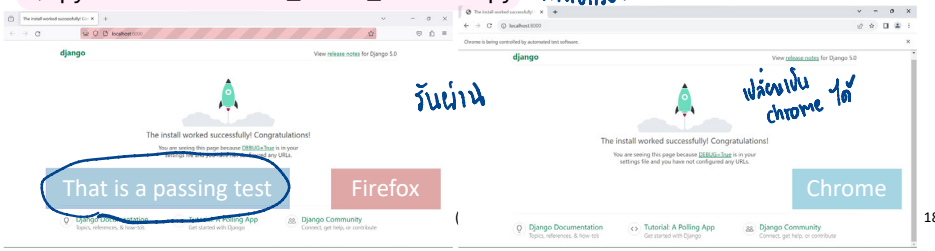
```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture/superlists
$ python manage.py runserver
```

- Open another command shell.

Do not forget to activate your virtualenv with workon superlists

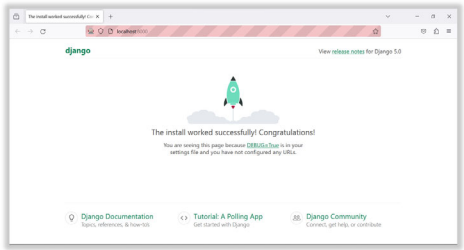
  - In that, we can try running our test again.

```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture/superlists
$ python functional_tests_firefox.py
```



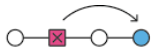
Getting Django Up and Running

- You can take a look at the dev server manually, by opening a web browser yourself and visiting http://localhost:8000 .
  - You should see something like this.
- You can quit the development server, and back in the original shell, using Ctrl-C .



Starting a Git Repository

- Commit our work to a version control system (VCS).
  - VCS tool can be used to
    - look back over old versions of code,
    - revert changes,
    - explore new ideas safely,
    - create a backup, etc.




- Let's start by moving *functional_tests_firefox.py* into the superlists folder, and doing the `git init` to start the repository:

```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture
$ mv functional_tests_firefox.py superlists/
```
ย้ายไปที่ /superlist

```
(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture
$ cd superlists/

(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture/superlists
$ git init .
Initialized empty Git repository in D:/somkiat/src/SE/Lecture/superlists/.git/

(superlists)
w_som@seiYoga7i14ITL5 MINGW64 /d/somkiat/src/SE/Lecture/superlists (master)
$ ls
db.sqlite3  functional_tests_firefox.py  manage.py*  superlists/
```
ใน superlists

สร้าง repo

Shell prompt will be simplified as $.

Take a look and see what files we want to commit:(

```
$ echo "db.sqlite3" >> .gitignore
$ echo "geckodriver.log" >> .gitignore
```
We don't want to track changes of **db.sqlite3** (a database file) and **geckodriver.log** (a logfile from Selenium) in version control.
We add both of them to a special file called .gitignore which tells Git what to ignore.

เพื่อไม่ให้ backup ไฟล์เหล่านี้


```
$ git add .
warning: in the working copy of '.gitignore', LF will be replaced by CRLF the
next time Git touches it
warning: in the working copy of 'functional_tests_firefox.py', LF will be
replaced by CRLF the next time Git touches it

$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   functional_tests_firefox.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/__pycache__/__init__.cpython-310.pyc
        new file:   superlists/__pycache__/settings.cpython-310.pyc
        new file:   superlists/__pycache__/urls.cpython-310.pyc
        new file:   superlists/__pycache__/wsgi.cpython-310.pyc
        new file:   superlists/asgi.py
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py
```
เพิ่มเข้าไปใน repo

It's pointless to commit **.pyc files**.

Let's **remove** them **from Git** and **add** them **to .gitignore** too

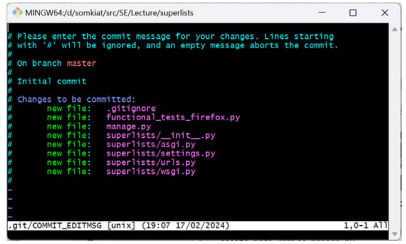

```
$ git rm -r --cached *.pyc
rm 'superlists/__pycache__/__init__.cpython-310.pyc'
rm 'superlists/__pycache__/settings.cpython-310.pyc'
rm 'superlists/__pycache__/urls.cpython-310.pyc'
rm 'superlists/__pycache__/wsgi.cpython-310.pyc'

$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore

$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   functional_tests_firefox.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/asgi.py
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .gitignore
```

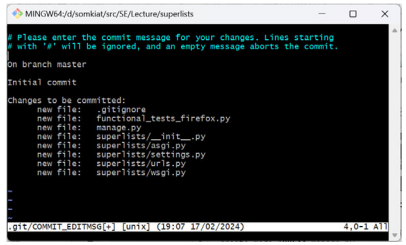
```
$ git add .gitignore
$ git commit
```

Let's do our first commit!

It will pop up an editor window for you to write your commit message in.

Edit the file as you want, save, and then close it.



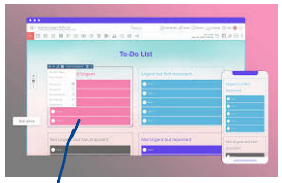



```
[master (root-commit) 441745d] On branch master
 8 files changed, 211 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 functional_tests_firefox.py
 create mode 100644 manage.py
 create mode 100644 superlists/__init__.py
 create mode 100644 superlists/asgi.py
 create mode 100644 superlists/settings.py
 create mode 100644 superlists/urls.py
 create mode 100644 superlists/wsgi.py
```

Test-Driven Development

Extending Our Functional Test Using the unittest Module

Extending Our Functional Test Using the unittest Module



- We are building "a to-do lists site".
  - The reason is that a to-do list is a really nice example.
  - At its most basic it is very simple indeed—just a list of text strings—so it's easy to get a "minimum viable" list app up and running.
  - It can be extended in all sorts of ways:
    - adding deadlines,
    - reminders,
    - sharing with other users, and
    - improving the client-side UI.

เพิ่มๆ ดู เอาแค่เป็น ที่เหลือไม่พอ

Extending Our Functional Test Using the unittest Module

- There's no reason to be limited to just "to-do" lists either.
  - They could be any kind of lists.
- But **the point** is that it should be used to **demonstrate** ...
  - all of the main aspects of web programming, and
  - **how to apply TDD** to them.

Using a Functional Test to Scope Out a Minimum Viable App

ใช้ตัวนี้ก่อน

- Tests that use **Selenium** let us **drive** a real web browser.
- So, they really let us see how the application **functions** from the user's point of view.
  - That's why they're called **functional tests**.

เป็นการทดสอบฟังก์ชันการทำงาน (แอพมาทำอะไรได้บ้าง)

- This means that an FT can be a sort of specification for your application.

ของว่า user จะใช้งาน ทำอะไรบ้าง

- It tends to track what you might call a User Story, and follows ...
  - how the user might work with a particular feature and
  - how the app should respond to them.

Functional Test == Acceptance Test == End-to-End Test == Black Box Test

Using a Functional Test

- FTs should have a human-readable story that we can follow.
- We make it explicit using **comments** that accompany the test code.
  - When creating a new FT, we can write the comments first, to capture the key points of the User Story.

  - Being human-readable, you could even **share** them with nonprogrammers, as a way of **discuss**ing the requirements and features of your app.

- TDD and agile software development methodologies often go together.

  → the **minimum viable app**.

  (the simplest thing we can build that is still useful)

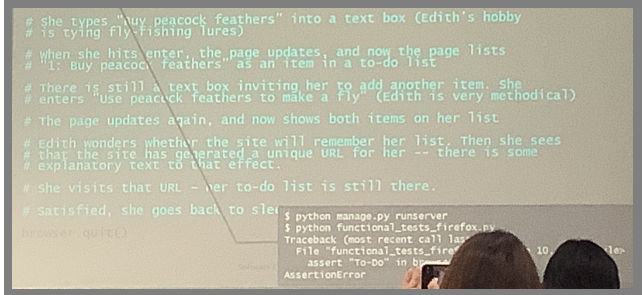
Using a Functional Test

- Let's start by building that.
- A minimum viable **to-do list** only needs to let …
  - the user enter some to-do items, and remember them for their next visit.
- Open up ***functional_tests.py*** and write a story a bit like this one:


```
from selenium import webdriver
from selenium.webdriver.firefox.options import Options

browser = webdriver.Firefox()

# Edith has heard about a cool new online to-do app. She goes
# to check out its homepage
browser.get("http://localhost:8000")

# She notices the page title and header mention to-do lists
assert "To-Do" in browser.title

browser.quit()
```

Updated the **assert** to look for the word "**To-Do**" instead of "Django"

Expected Fail !!!



A Word for Comments…

- It's pointless to write a comment that just repeats what you're doing with the code:

```
# increment wibble by 1
wibble += 1
```

- Not only is it pointless, but there's a danger that you'll forget to update the comments when you update the code, and they end up being misleading.
- The ideal is to strive to make your code so readable, to use such good variable names and function names, and to structure it so well that …
  - You no longer need any comments to explain what the code is doing.
  - Just a few here and there to explain why.
- Here comments are used to explain the User Story in our functional tests—by forcing us to make a coherent story out of the test, it makes sure we're always **testing from** the point of **view of the user**.

The Python Standard Library's unittest Module

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get('http://localhost:8000')

        # She notices the page title and header mention to-do lists
        self.assertIn('To-Do', self.browser.title)
        self.fail('Finish the test!')

        # She is invited to enter a to-do item straight away
        #[...rest of comments as before]
```

Tests are organised into **classes**, which **inherit** from `unittest.TestCase`.

The **main body** of the **test** is in a method called **`test_can_start_a_list_and_retrieve_it_later`**.
Any method whose name starts with test is a test method, and will be run by the test runner.
You can have more than one test_ method per class.
Nice descriptive names for our test methods are a good idea too.

The Python Standard Library's unittest Module

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get('http://localhost:8000')

        # She notices the page title and header mention to-do lists
        self.assertIn('To-Do', self.browser.title)
        self.fail('Finish the test!')

        # She is invited to enter a to-do item straight away
        #[...rest of comments as before]
```

`setUp` and `tearDown` are special methods which get **run before** and **after each test**.

They are used here to start and stop our **browser**.
No more Firefox **windows left** lying around!

`self.fail` just fails no matter what, producing the error message given.
It is used here as **a reminder to finish the test.**

Use `self.assertIn` instead of just assert to make our test assertions.
`unittest` provides **lots of helper functions** like this to make test assertions, like **`assertEqual`**, **`assertTrue`**, **`assertFalse`**, and so on.

The Python Standard Library's unittest Module

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get('http://localhost:8000')

        # She notices the page title and header mention to-do lists
        self.assertIn('To-Do', self.browser.title)
        self.fail('Finish the test!')

        # She is invited to enter a to-do item straight away
        #[...rest of comments as before]

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

The `if __name__ == '__main__'` clause is how a Python script **checks if** it's been **executed from the command line**, rather than just imported by another script.

We call `unittest.main()`, which **launches** the **unittest test runner**, which will **automatically find test classes** and **methods** in the file and **run them**.

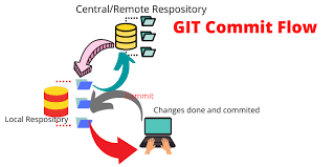
The Python Standard Library's unittest Module

```
$ python functional_tests_firefox.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\somkiat\src\SE\Lecture\superlists\functional_tests_firefox.py",
line 18, in test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'The install worked successfully!
Congratulations!'

----------------------------------------------------------------------
Ran 1 test in 12.312s

FAILED (failures=1)
```

The `assertIn` has given us a **helpful error message** with useful debugging info.

It gives us a nicely **formatted report** of **how many tests** were run and **how many failed**.

Commit



- This is a good point to do a commit; it's a nicely self-contained change.
  - We've expanded our functional test to include comments that describe the task we're setting ourselves, our minimum viable to-do list.
  - We've also rewritten it to use the Python unittest module and its various testing helper functions.


```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   functional_tests_firefox.py

no changes added to commit (use "git add" and/or "git commit -a")
```

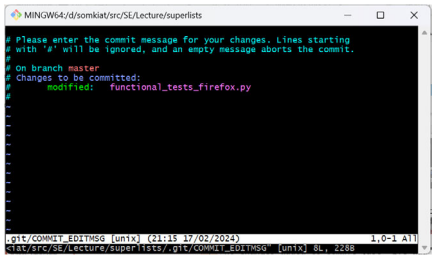
To assure that the **only file that has changed** is `functional_tests_firefox.py`

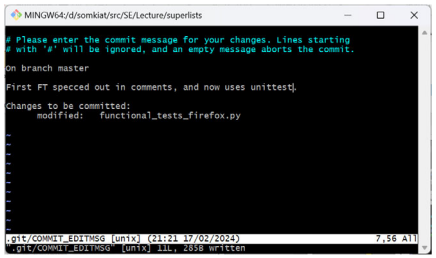
```
$ git diff
warning: in the working copy of 'functional_tests_firefox.py', LF will be
replaced by CRLF the next time Git touches it
diff --git a/functional_tests_firefox.py b/functional_tests_firefox.py
index cc82d40..cea22b9 100644
--- a/functional_tests_firefox.py
+++ b/functional_tests_firefox.py
@@ -1,8 +1,48 @@
 from selenium import webdriver
+import unittest

-browser = webdriver.Firefox()
+class NewVisitorTest(unittest.TestCase):
+
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+
+    def tearDown(self):
[...]
```

Show the **difference** between the **last commit** and **what's currently on disk**

```
$ git commit -a
```

The -a means "**automatically add any changes to tracked files**"
(i.e., any files that we've committed before).
It **won't add any brand new files** (you have to explicitly git add them yourself), but often, as in this case, there aren't any new files, so it's a useful shortcut.





When the editor pops up,
add a descriptive commit message,
like "First FT specced out in comments, and now uses unittest."

```
warning: in the working copy of 'functional_tests_firefox.py', LF will be
replaced by CRLF the next time Git touches it
[master c08183f] On branch master
 1 file changed, 44 insertions(+), 4 deletions(-)
```

Now start writing some real code for our lists app.