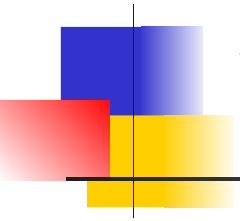


Web Application Testing in Python

With an Intro to Selenium WebDriver

Guidance

What to Test?

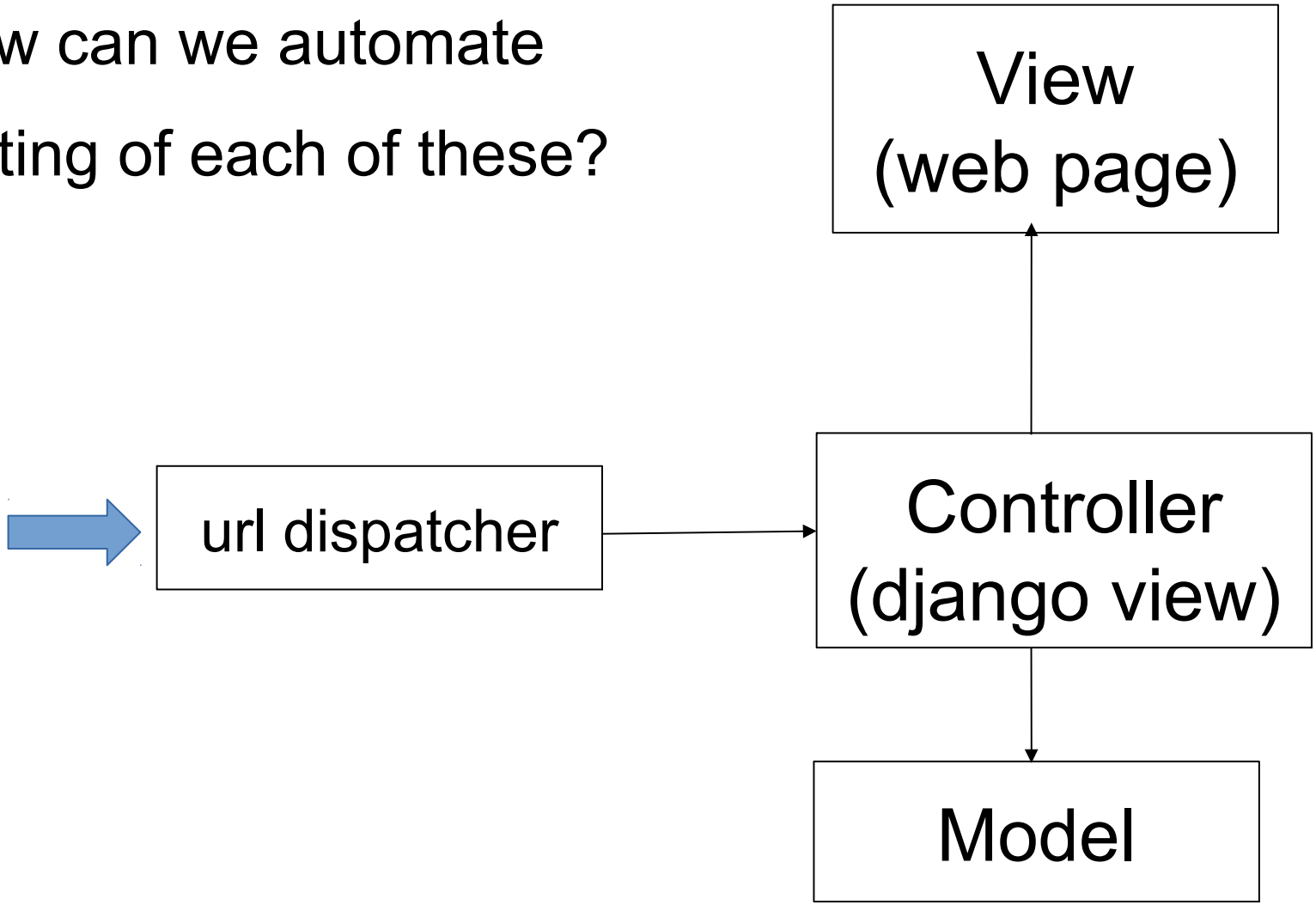
- Application conforms to the requirements
- Logic
- Flow Control
- Application Flow, e.g. Page Flow
- Configuration

"Don't test Constants", e.g. HTML template text

Test your code, not the framework

Testing Parts of a Web App

How can we automate testing of each of these?



Models: test using standard unit tests

```
import django.test
from polls.models import Question

class QuestionTest(django.test.TestCase):
    def test_question_with_future_date(self):
        tomorrow = timezone.now() +
                datetime.timedelta(days=1)
        question = Question( question_text=
          "Is this the future?", pub_date=tomorrow)
        # future date is not "recent"
        self.assertFalse(
                question.was_published_recently() )
```

Not necessary to test the *framework*

```
import django.test
from polls.models import Question

class QuestionTest(django.test.TestCase):
    def setUp(self):
        Question.objects.create(
                question_text="Question One")
        Question.objects.create(
                question_text="Question Two")

    def test_create_questions(self):
        self.assertEqual(2, Questions.objects.count())
```

What is Being Tested? (cont'd)

```
    def test_question_text(self):
        self.assertTrue(
            any("Question One" in q.question_text
                for q in Questions.objects.all() )
```

This is testing Django's persistence framework.

OK to do it occasionally while learning Django.

But its not a useful test of your code.

Django Views and URLs

Use `django.test.Client` to experiment

```
$ python manage.py shell
>>> from django.test import Client
>>> c = Client()
# Get the /polls/ page. Should contain some polls
>>> response = c.get('/polls/')
# Did it succeed?
>>> response.status_code
200
# Print the html content
>>> response.content
'<html>\n<head>\n<style>...\n<h1>Active Polls</h1>...
```

Testing Django Views and URLs

```
class TestViews(django.test.TestCase):
    def setUp(self):
        self.client = django.test.Client()

    def test_polls_index(self):
        poll = Question(question_text="ABCDEFGHIJ",...)
        poll.save()
        response = self.client.get('/polls/')
        self.assertEqual(response.status_code, 200)
        # Is test poll included in the page?
        self.assertContains(response, "ABCDEFGHIJ")
```

Rewrite the Test using reverse()

Instead of writing "`/polls/`" URL as a String, use reverse() to get the URL by name from `urls.py`.

```
from django.shortcuts import reverse


def test_polls_index(self):
        poll = Question(question_text="ABCDEFGHIJ",...)
        poll.save()
        url = reverse('polls:index')
        response = self.client.get( url )
        self.assertEqual(response.status_code, 200)
        # Is test poll included in the page?
        self.assertContains(response, "ABCDEFGHIJ")
```

Test the / URL is Redirected

Test that 'GET /' redirects the browser to polls index.

```
def test_redirect_root_url(self):
    """root url should redirect to polls index"""
    response = self.client.get('/')
    # Test using the basic way
    self.assertEqual(response.status_code, 302)
    polls_url = reverse('polls:index')
    self.assertEqual(response.url, polls_url)

    # Better way: use TestCase assertRedirects
    self.assertRedirects(response, polls_url)
```

Explore Tests using Django Shell

If you are not sure how to test, use Django Shell to try it

```
>>> tc = django.test.TestCase()
>>> client = django.test.Client()
# The root url / should redirect to polls
>>> response = client.get('/')
>>> tc.assertRedirects(response, '/polls/')
# "Location" header field is the redirect url
>>> assert response.get('Location') == '/polls/'
```

Useful django.test.TestCase asserts

```
# the response contains some text

assertContains( response, "some text")

assertNotContains( response, "bad text")

# response is a redirect to some url

assertRedirects( response, url )

#  reponse uses the template we expect

assertUsesTemplate( response,
                    'polls/detail.html')
```

Useful Info in HttpResponse

When you invoke client.get() or client.post() the `response` in an HttpResponse containing these fields:

`status_code` - the HTTP status code (200 = OK, etc.)

`request` - the HttpRequest that caused this response

`templates` - a list of templates used in the response

`content` - body of the response, as a byte-string

`context` - the context that was used to render the template; context contains a key-value map.

Where is the info for TestCase?

How do you know what TestCase and Client can do?

See the "Django Testing Tools" page.

`https://docs.djangoproject.com/en/3.1/topics/testing/tools/`

The asserts are buried near the bottom of the page, in section "Assertions".

How to Test a Template?

```
def test_polls_index(self):
      poll = Question(question_text="ABCDEFGHIJ",...)
      poll.save()
      response = self.client.get('/polls/')
      self.assertTemplateUsed(response,
            template_name='polls/index.html')
```

You can test POST, too

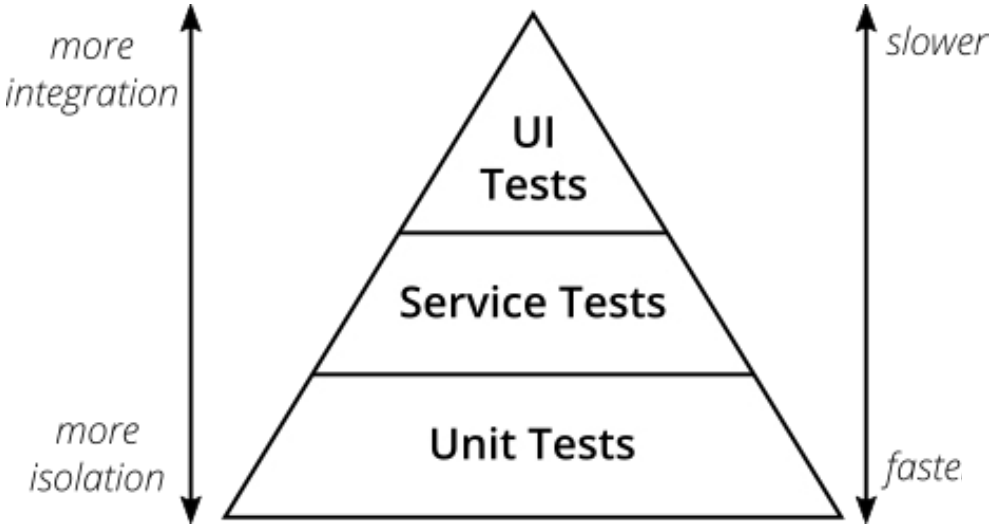
```
# Vote for a poll.
# This example assumes you somehow know the poll
# id is 1.  Send POST data as a Python dictionary.
response =
    self.client.post('/polls/1/', {'choice': '2'})
# What should POST return? (Should redirect)
# Test that the vote was recorded in choice.
# Test an invalid choice
response2 =
    self.client.post('/polls/1/', {'choice': '9999'})
```

Are Unit Tests Enough?

No.

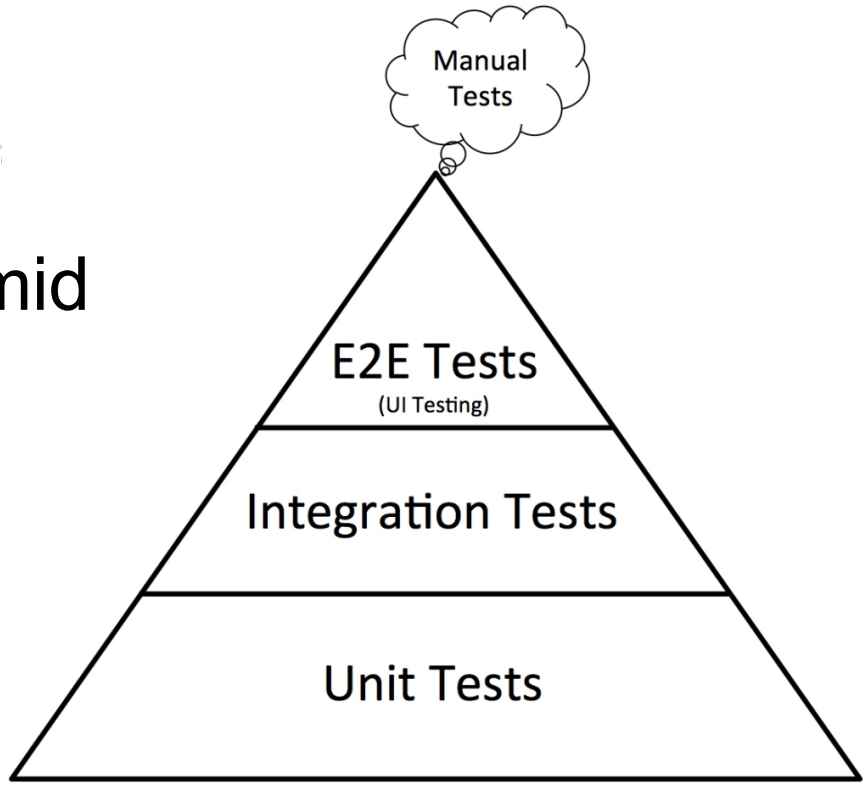
Unit tests don't test that the application works.

The Testing Pyramid



Mike Cohen's original Pyramid

"Practical" pyramid



Integration Testing

Test the interaction between components.

- Components belonging to your app
- Back-end services called by front-end
- External components and web services

Examples:

- database
- file system used to save user uploaded files
- a Google API used by your app

How to Test:

- often access a "service layer" or your standard URLs.

Functional or "End-to-End" Tests

Test the "development" or "production" app while its running! -- not a 'test' server.

Run tests through an actual web browser.

Test the application as a whole.

Goals of E2E Testing

1. Verify the application works "*from start to finish*".
2. Can perform the major use cases (user stories) via the browser interface.

   - called "happy path" (nothing goes wrong)

3. Test that required services work, too.

Goal is not comprehensive testing of every feature.

Secondary Use of E2E Tests

Useful to verify all links, buttons, and menu items work.

No "404" or "5xx" errors.

E2E Testing Tools

Selenium - control an actual web browser using code.

- Interface in many languages, incl. Python & Java
- Django has built-in support
- Selenium IDE for creating tests in a web browser

Cypress.io - Javascript testing tool. Natively interacts with pages in your application.

- uses Mocha and Chai for writing tests
- tests written in Javascript

Puppeteer - library for controlling a "headless" Chrome browser. Uses Javascript and node.js.

- uses: page scraping, web crawling, testing

Selenium

Browser automation. Not just testing.

`https://selenium.dev/`

We will use Selenium WebDriver

- programmatically control a web browser

Selenium Example

Goal:

Use duckduckgo.com to find links to Kasertsart U.

Print the top 10 links.

Requires:

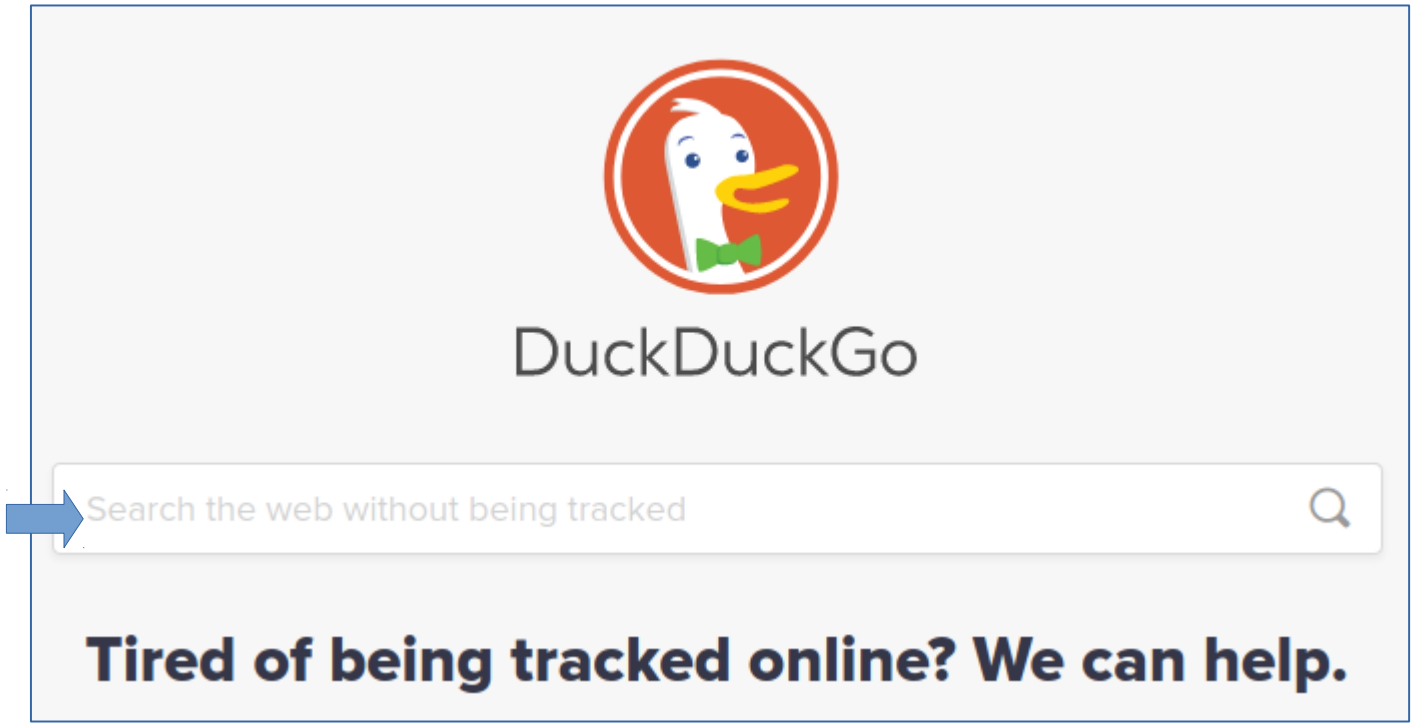
- Selenium WebDriver (`pip install selenium`)
- driver for Firefox browser (called "geckodriver")

  https://github.com/mozilla/geckodriver/releases
- you can use Chrome or Safari instead

Selenium: get a web page

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
# browser: a WebDriver object
browser = webdriver.Firefox()
browser.implicitly_wait(5) # seconds


# get the duckduckgo search page
url = "https://duckduckgo.com"
browser.get( url )
```

Get the id of the search input box



Firefox: right-click in search box -> "Inspect Element".

You will see:

<input id='**search_form_input_homepage**' name="**q**"
type="text" ...>

find the id on page & send data

```
# Find the search box on page
# Selenium has many find_by_* commands

field_id = 'search_form_input_homepage'

input_field =
     browser.find_element_by_id(field_id)

input_field.send_keys("Kasetsart Univer")

input_field.send_keys(Keys.ENTER)

# Run It!

# the browser should display results
```

Inspect the Page & Identify Links

We need a way for Selenium to "find" the hyperlinks on the results page.

You can use:

* tag type ('a' tag)

* "id" of an element

* "class" of an element

* CSS selectors, or other identifying data

```
<div class="...">
  <a class="result__url js-result-extras-url"
   href="https://www.usnews.com/education/
   best-global-universities/kasetsart-university-xxx"
   ...>
```

Page Scraping

```
# get links from the results page
# the link URLs have class="result__url"

elements =
  browser.find_elements_by_class_name(
  'result__url')

print(f"Found {len(elements)} matches.")

# Each result is a WebElement object.
# WebElement contains attributes &
# other (child) WebElements.
# Show "href" attribute of first match

url = elements[0].get_attribute('href')
```

Page Scraping (2)

```
# Get the 'href' value

page_url =
  elements[0].get_attribute('href')

print("First result link is", page_url)

# What the heck -- Let's visit the page!

# element must be "clickable" to work

elements[0].click()


input("Press ENTER to go back to results")

browser.back()
```

Exercise: print first 10 URLs

1. Print the URLs of the first 10 matches on the DuckDuckGo search results page.

2. Make this code into a function that you can use to get (and return) the top-10 results for any search!

```
elements =
  browser.find_elements_by_class_name(
   'result__url')

# TODO print the first 10 result URLs
```

Another Way to Find Links

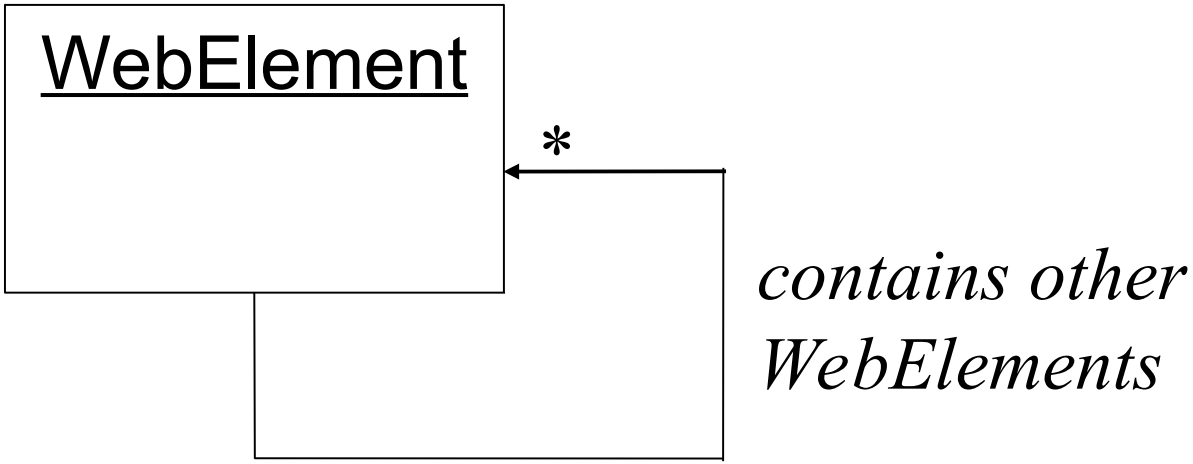
```
# The Hyperlinks use class 'result__a'

links = browser.
  find_elements_by_class_name('result__a')

for link in links:

    if link.tag_name == 'a':

        url = link.get_attribute('href')

        print(url)
```

Composite Design Pattern

`WebElement` may contain other `WebElements`.

`WebElement` is the primary object for interacting with a web page using Selenium.

`WebDriver` contains many of the same methods as WebElement



Headless Browsing

You can run a browser without opening a U.I. window.

This is called headless mode.

May be necessary when running E2E tests on a C.I. server.

It is *faster*, too.

https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Headless_mode

References

The Practical Test Pyramid

https://martinfowler.com/articles/practical-test-pyramid.html

Good Selenium Tutorial in Python (7 parts)

https://blog.testproject.io/2019/07/16/set-your-test-automation-goals/

The same author has other good testing tutorials:

https://blog.testproject.io/2019/07/16/

Django E2E Tests with Selenium

TDD in Python (online book)

Several chapters use Selenium for E2E testing of the Django project used in book.

Testing the Github Public API

developer.github.com/v3/users/