Django Review

Python Syntax used in Django

1. Lists

2. Dictionaries

3. Key-word arguments (**kwargs)

Python lists

Python list syntax looks like an array.

```
> fruit = [ "apple", 'banana', "orange" ]
> len(fruit)            # invokes fruit.__len__()
3
> fruit[1]
'banana'
> fruit[1] = "mango" # change fruit[1] to mango
[ "apple", 'mango', "orange" ]
> fruit.pop()           # remove last element & return
"orange"
> fruit                 # pop() removed last element
['apple', 'mango']
> fruit.append('fig')
```

Python dictionary

A **key-value** collection, like `Map` in Java.

```
> langs = {"python":"easy", "java":"cool"}
> langs.keys()     # order is not preserved
dict.keys(['java', 'python'])
> langs['java']
'cool'
> langs['ruby'] = "looks like Perl"
> for lang in langs:  # iterate over all keys
    print("{0} is {1}".format(lang, langs[lang]))

ruby is looks like Perl
java is cool
python is easy
```

**kwargs

`**kwargs` is a *dictionary* of named arguments (**k**ey **w**ord **args**) and values. The names can be anything.

You can use any name for the "`kwargs`" parameter.

The help for many Django methods looks like this:

```
Question.objects.create(*args, **kwargs)

poll = Question.objects.create(
   name="Who will be next U.S. president?",
   pub_date=timezone.now()
   )
```

**kwargs must be the last parameter

It should be the last parameter in a function signature.

```
def myfun(x, **kwargs):
    print("x=", x)    # required parameter
    print("Optional arguments:")
    for key in kwargs:
        print(key, "=", kwargs[key] )

myfun("hi", id=219245, name="ISP", size=37)
```

Django Page Templates

In a **template**, you put *variables* inside {{ ... }}

templates/polls/details.html:

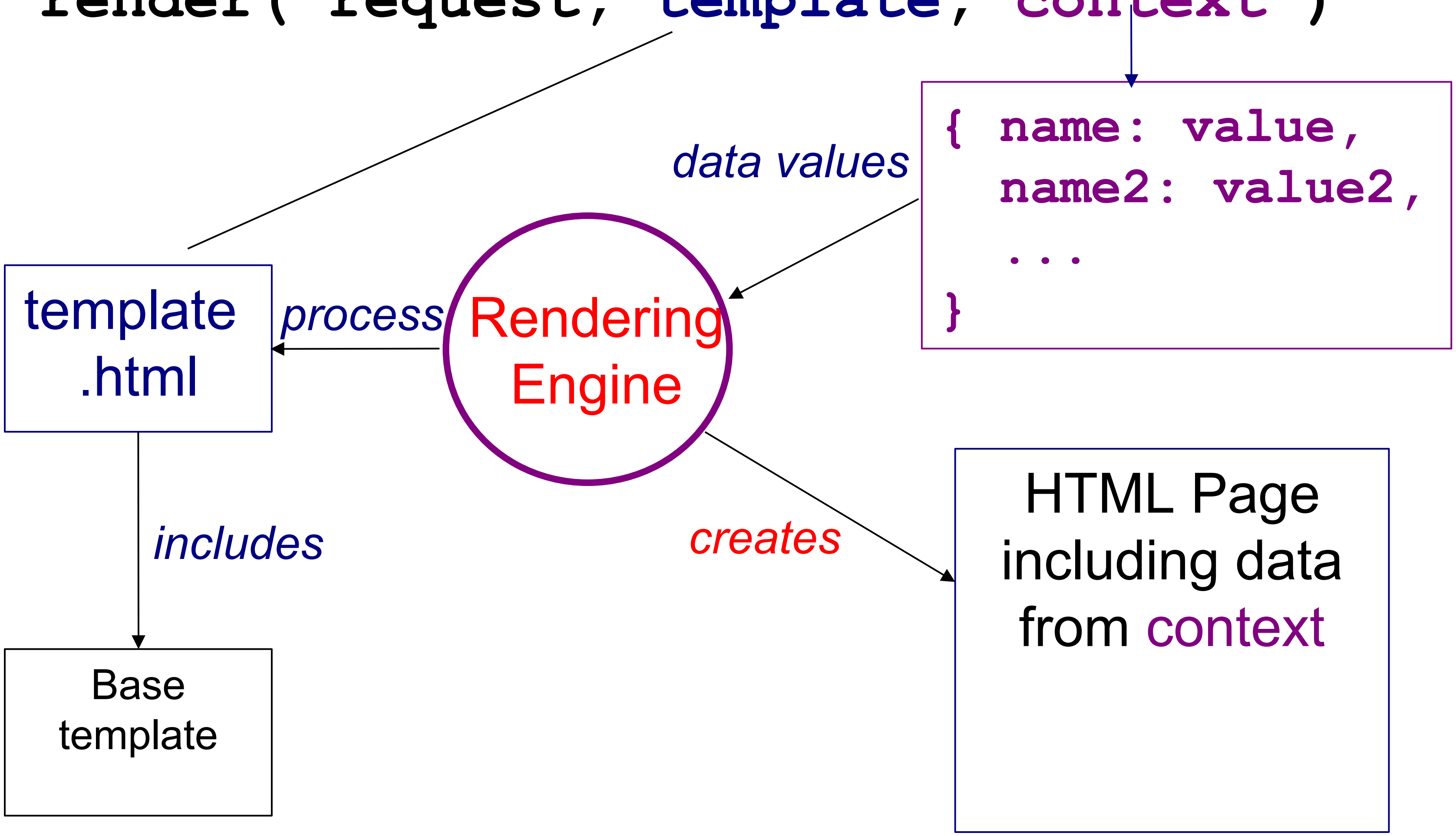
```
<p>
Q{{question.id}} is "{{question.question_text}}"
</p>
<!-- a template can invoke a method, too -->
{{question.was_published_recently}}
```

```
Q1 is "What is your favorite food?"
True
```

Rendering a Template

A "rendering engine" processes the template.

```
render( request, template, context )
```



Python code for rendering

In a view method:

```
from django.template import loader
template =
    loader.get_template('polls/details.html')

# context = key-values to use in template
context = {'question': question, ...}
html = template.render(context, request)

return HttpResponse(html)
```

Shortcut for rendering

```
from django.shortcuts import render

context = {'question': question, ...}

# render returns an HttpResponse object
return render(request,
              'template.html', context)
```

Can also access `request` data

A **template** can also access vars from the `request`.

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.get_username }}.</p>
{% else %}
    <p>Welcome, web surfer.</p>
{% endif %}
```

`user` refers to `request.user`

`user.get_username` refers to
  `request.user.get_username()`

Code Should be Easy to Read

Instead of:

```
return render(request,'template.html',
     {'question': question, ...} )
```

add ***explanatory variable***

```
context = {'question': question,...}
return render(request, 'template.html',
            context )
```

In a "view" what is request?

A Django "view" function looks like this:

```
from django.http import HttpResponse
from django.template import loader

def detail(request, question_id):
    questions = Question.objects.all()[0:10]
    context = {'question_list':questions}
    template = \
            loader.get_template('some_file')

    return HttpResponse(
        template.render(context, request ) )
```

What is HttpResponse?

What does `HttpResponse represent`?

```
from django.http import HttpResponse
from django.template import loader

def detail(request, question_id):
    questions = Question.objects.all()[0:10]
    context = {'question_list':questions}
    template = \
             loader.get_template('some_file')

    return HttpResponse(
        template.render(context, request ) )
```

URL Dispatching

Each "app" can have a `urls.py` to match request URLs and dispatch them to a "view".

```
from django.urls import path

# app_name is used to define a namespace
# (used for "reverse mapping")
app_name = 'polls'

url_patterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/',
                  views.detail, name='detail'),
    path('<int:question_id>/vote/',
                  views.vote, name='vote'),
    path('<int:question_id>/results/',
                  views.results, name='results'),
    ]
```

Dispatch these URLs

Which view would handle each of these requests:

1) http://localhost:8000/polls/
2) http://localhost:8000/polls/4/
3) http://localhost:8000/polls/8/vote?username=nok
4) http://localhost:8000/polls/8/vote/summary

```
# URL mapping for /polls/ app
url_patterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/',
             views.detail, name='detail'),
    path('<int:question_id>/vote/',
             views.vote, name='vote'),
    ]
```

Mapping from View to URL

Inside html template, we want to insert a URL of a view.

Example: add a link to the polls index page.

How to "build" this URL inside a template?

```
<!-- question details template -->
<p>Question Id: {{ question.id }} </p>
<p>Text: {{ question.question_text }} </p>
<a href="/polls/index">Back to Polls index</a>

<a href="{% url 'polls:index' %}">
    Back to Polls index
</a>
          app_name                view name
```

>> Notice that {%...%} is processed even inside "..."

Why is creating URL for a view important?

Reverse Dispatch

Sometimes a view controller wants to <u>redirect</u> the user to a different URL.

```
from django.http import HttpResponseRedirect

def vote(request,question_id):
    question = Question.objects.get(id=question_id)
    // TODO save the vote for this question
    ...
    // Show all votes for this question
    _____Redirect to polls/{id}/results_________
    return ???
```

How to <u>redirect</u> the browser to this page?

Reverse Dispatch: reverse()

Redirect uses info from the urls.py files to construct the URL the user should go to.

```
from django.http import HttpResponseRedirect

def vote(request,question_id):
    q = Question.objects.get(id=question_id)
    ## TODO get user's choice and add +1 to votes
    ...
    # Redirect browser to page of vote results
    HttpResponseRedirect(
         reverse('polls:results',args=(q.id,) ) )
```

Get the URL that matches the named route

Thorough Testing is Needed!

Python code is *interpretted*.

There is no pre-compilation to catch errors (as in Java). So, you need to **test every path of execution**.

```
NameError at /polls/1/vote/
name 'reverse' is not defined
```

Programmer forgot (in views.py):

```
from django.urls import reverse
```

but error is not detected until `reverse`() is encountered at run-time.

All Frameworks must do this

Most web apps need a way to:

1. Include links to other app URLs in an HTML page
   - Amazon products page has links to each product
2. Redirect user to another page in our app
   - After add item to cart, redirect to view_cart page.

Issue:

How to *inject* the <u>correct</u> URLs, without hardcoding them?

Django's Solution

Most web apps need a way to:

1. Include link to other URLs **in an HTML template**

   `{% url 'app_name:view_name' args %}`

2. Redirect user to another page **in a view**

   ```
   HttpResponseRedirect(
        reverse('app_name:view_name',
       args=(...)))
   ```

Rationale:

Make "apps" reusable by providing a naming of URL mappings at the app level, e.g. "`polls:results`".

GET and POST

GET is used to request a web resource, such as a web page.

GET /polls/1/

What is POST used for?

(*Semantic meaning of POST*)

1. Send data to the application, such as from a form.

Your name: <input type="text" name="username" />

<p>some text</p>

<br />

2. To create a resource on the server.

One view for <u>both</u> GET and POST

One view can handle both.

Use **`request.method`** to determine which method.

```
def detail(request, question_id):
    question = Question.objects.get(id=question_id)

    if request.method == 'GET':
        # render and return the details template

    elif request.method == 'POST':
        # handle user's vote
        choice = request.POST['choice']

        # after a POST, always redirect somewhere
        return redirect('polls:results', args=(...))
```

Exploring Models

Use Django to start an interactive Python shell.

This is described in Tutorial part 2.

```
python manage.py shell  [ -i python ]

>>> from polls.models import Question, Choice
>>> q = Question.objects.get(id=1)
>>> q.question_text
"What is your favorite programming language?"
>>> choices = q.choice_set.all( )
>>> for c in choices:
...     print("%-10s %d" % (c.choice_text, c.votes))
Basic      0
C          1
Java       4
Python     2
```

Try out Persistence

Try persistence operations: save(), get(), delete()

```
>>> c = Choice()
>>> c.choice_text = "Lisp"  # or "Racket" ("Scheme")
>>> c.votes = 2
## Foreign Key.  You have to find this separately.
>>> c.question_id = 1
>>> c.save()
>>> for choice in q.choice_set.all():
...     print(choice)
## Now the output includes "Lisp"
>>>
```

Persistence Operations: CRUD

All Persistence Frameworks provide a way to...

- Create (save) an entity to the database
- Retrieve an object, by id or by field value (query)
- retrieve all objects
- Update object data in database
- Delete an entity (object) from database

How does Django do these?

Testing

Django Unit Tests extend TestCase class.

```
public class QuestionModelTest(TestCase):
    def test_create_question(self):
        question = Question(question_text="this is a test")
        self.assert
```

Wrong Name!

In Tutorial, name is "`QuestionModel`**`Tests`**".

It should be "`xxxTest`" (no "s")!

Don't use plural for your test classes.

What is a django.test.TestCase ?

```
>>> from django.test import TestCase
>>> help(TestCase)
class TestCase(TransactionTestCase)
    ...
    Method resolution order:
        TestCase
        TransactionTestCase
        SimpleTestCase
        unittest.case.TestCase
        builtins.object
```

Running Tests

```
cmd>  python manage.py test polls
```

Criticisms:

- Django test code is in same directory as production code.
- Should have separate "test" files for each target, don't bundle them into one file `(tests.py)`
- `tests.py` is poor name. Test what? Don't use plural (no "s")!

Design: Low Coupling

Good software design strives for low coupling.

Especially, low or no coupling between unrelated parts.

What features of Django reduce coupling?

1. Django divides a project into self-contained "apps"

2. {% url 'name' %} reduces coupling between URLS and templates
3. ???

Design: Portability and Reuse

Good software design enables portability and code reuse.

A framework itself is both portable and reusable (we use it to create our own web app)!

How does Django enable us to move or reuse our own web application code?

Django and Git

When you commit your Django project to Git,
what files should you **not commit**?

> Add them to `.gitignore`

> If you don't know what to put in .gitignore, create a repo on Github and ask Github to create a `.gitignore` file for you.

> What is `*.pyc` ? What is `*.py[cod]` ?

Is Django a Web Server?

[ ] Yes

[ ] No

Django is Not a Web Server

But I can type: `manage.py runserver`

and it works *right out of the box*.
How to you explain ***that***?

Web Developer

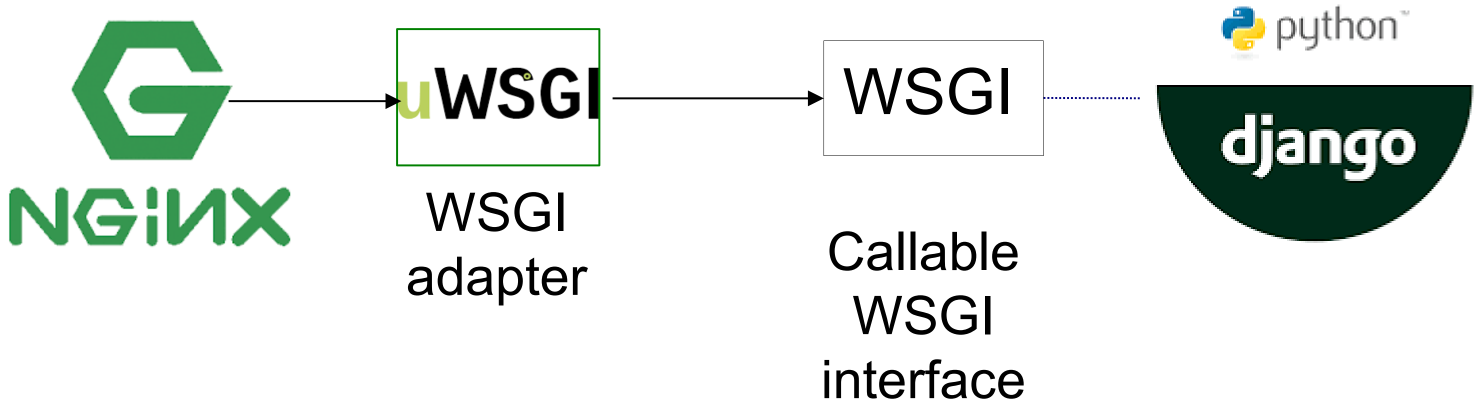
Django includes a "light-weight" HTTP server

Intended for development only.

Not suitable for production (Tutorial, part 1).

Django uses WSGI interface

WSGI (Web Server Gateway Interface) is a standard interface for *communication* between a Python web app and a web server.



You can run Django in any web server that:

- supports WSGI or has an *adapter* for WSGI interface