Lecture 10

[Web [Application]] Frameworks
Conventional approach to building a web site

- user interface, logic, database access are all mixed together

```python
import MySQLdb
print "Content-Type: text/html"
print
print "\n<html><head><title>Books</title></head>"
print "\n<body>"
print "\n<h1>Books</h1>"
print "\n<ul>"
connection = MySQLdb.connect(user='me', passwd='x', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC")
for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]
print "\n</ul>"
print "\n</body></html>"
connection.close()
```
Overview of web application frameworks

• client-server relationship is stereotypical
  – client sends requests using information from forms
  – server parses request, calls proper function, which retrieves from database, formats response, returns it

• REST: URL filenames often used to encode requests
  …/login/name
  …/add/data_to_be_added
  …/delete/id_to_delete

• server uses URL pattern to call proper function with proper arguments

• server usually provides structured & safer access to database
• server may provide a templating language for generating HTML
  – e.g., replace { % foo % } with value of variable foo, etc.
• framework may automatically generate an administrative interface
• often provides library routines for user ids, passwords, registration, etc.
Flask: Python-based microframework

```python
$ cat hello.py
    import flask
    app = flask.Flask(__name__)
    @app.route('/
    def hello():
        return 'Hello world'
    app.run()

$ python hello.py

Hello world
```
Sending form data

<html>
<title>Survey demo</title>
<body>
<form METHOD=POST ACTION="http://localhost:5000">
<p> Name: <input type="text" name=Name id=Name>
<p> Netid: <input type="text" name=Netid id=Netid>
<p> Class:
    <input type="radio" name=Class value="2019"> '19
    <input type="radio" name=Class value="2020"> '20
    <input type="radio" name=Class value="2021"> '21
<p> Courses:
    <input type="checkbox" name=C126> 126
    <input type="checkbox" name=C217> 217
    <input type="checkbox" name=C226> 226
<p> <input type="submit" value="Submit"> <input type=reset>
</form>
</body>
</html>
Processing form data

# survey.py

from flask import Flask, request

app = Flask(__name__)

@app.route('/', methods=['POST','GET'])
def survey():
    s = ""
    for (k,v) in request.form.iteritems():
        s = "%s %s=%s<br>" % (s, k, v)
    return 'Form contents:<br>' + s

app.run()
Flask interaction

Request:

Name: joe college
Netid: joec

Class: '19 □ '20 □ '21
Courses: □ 126 □ 217 □ 226
Submit  Reset

Response:

Form contents:
Netid=joec
C126=on
Class=2019
Name=joe college
Python @ decorators

- a way to insert or modify code in functions and classes
  ```python
  @decorate
  def foo(): ...
  ```
- compilation compiles foo, passes the object to decorate, which does something and replaces foo by the result
- used in Flask to manage URL routing

```python
@app.route('/find/<str>')
def find(str):
    all = query_db('select * from lines')
    s = ''
    for i in all:
        if re.search(str, i[0]) != None:
            s = "%s%s
            " % (s, i[0])
    return '<pre>%s
%s</pre>' % (str, s)
```
Django: more heavyweight Python-based framework

- by Adrian Holovaty and Jacob Kaplan-Moss (released July 2005)

- a collection of Python scripts to
  - create a new project / site
    - generates Python scripts for settings, etc.
    - configuration info stored as Python lists
  - create a new application within a project
    - generates scaffolding/framework for models, views
  - run a development web server for local testing

- generate a database or build interface to an existing database
- provide a command-line interface to application
- create an administrative interface for the database
- run automated tests
- ...

Django Reinhart, 1910-1953
Model-View-Controller (MVC) pattern

- **model**: the structure of the data
  - how data is defined and accessed

- **view**: the user interface
  - what it looks like on the screen
  - can have multiple views for one model

- **controller**: how information is moved around
  - processing events, gathering and processing data,
    generating HTML, ...

- separate model from view from processing so that when one part changes, the others need not

- used with varying fidelity in frameworks

- not always clear where to draw the lines
  - but trying to separate concerns is good
**Django web framework**

- **write client code in HTML, CSS, Javascript, ...**
  - Django template language helps to separate form from content
- **write server code in Python**
  - some of this is generated for you
- **write database access with Python library calls**
  - they are translated to SQL database commands

- **URLs on web page map mechanically to Python function calls**
  - regular expressions specify classes of URLs
  - URL received by server is matched against regular expressions
  - if a match is found, that identifies function to be called and arguments to be provided to the function
Django automatically-generated files

- **generate framework/skeleton of code by program**

- **three basic files:**
  - models.py: database tables, etc.
  - views.py: business logic, formatting of output
  - urls.py: linkage between web requests and view functions

- **plus others for special purposes:**
  - settings.py: db type, names of modules, ...
  - tests.py: test files
  - admin.py: admin info
  - templates: for generating and filling HTML info
Example database linkage

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': '/Users/bwk/django/sql3.db', ...
    }
}

from django.db import models

class Post(models.Model):
    title = models.TextField(5)
    text = models.TextField()

BEGIN;
CREATE TABLE "blog_post" (  
    "id" integer NOT NULL PRIMARY KEY,
    "title" text NOT NULL,
    "text" text NOT NULL
);

in settings.py
in models.py
generated by Django
URL patterns

- regular expressions used to recognize parameters and pass them to Python functions
- provides linkage between web page and what functions are called for semantic actions

```python
urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/\d{1,2}$/', hours_ahead),
)
```

- a reference to web page .../time/ calls the function
  ```python
current_datetime()
```

- tagged regular expressions for parameters: url .../time/plus/12 calls the function
  ```python
  hours_ahead(12)
  ```
Templates for generating HTML

• try to separate page design from code that generates it

• Django has a specialized language for including HTML within code
  – loosely analogous to PHP mechanism

# latest_posts.html (the template)

<html><head><title>Latest Posts</title></head><body>
<h1>Posts</h1>
<ul>
{% for post in post_list %}
  <li>{{ post.title }} {{ post.text }}</li>
{% endfor %}
</ul>
</body></html>
**Administrative interface**

- most systems need a way to modify the database even if initially created from bulk data
  - add / remove users, set passwords, ...
  - add / remove records
  - fix contents of records
  - ...
- often requires special code

- Django generates an administrative interface automatically
  - loosely equivalent to MyPhpAdmin
Alternatives…

- Ruby on Rails
- Google App Engine
- Node + Express
- and lots of others
Node.js server

```javascript
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World
');
}).listen(1337, '127.0.0.1');
```

- Express framework for Node
  - analogous to Flask
var express = require('express')
var app = express()

app.get('/', function (req, res) {
    res.send('Hello World!')
})

app.listen(3000, function () {
    console.log('Example app listening on port 3000!')
})
Package managers

• **pip**  
  Python (pypi.python.org/pypi/pip)
  ```
  pip install Django
  ```

• **apt-get**  
  Ubuntu Linux
  ```
  apt-get install whatever
  ```

• **npm**  
  Node.js (yarn is a wrapper around npm)
  ```
  npm install node
  ```

• **port**  
  Macports
  ```
  port install ruby
  ```

• **brew**  
  Homebrew
  ```
  brew install ruby
  ```

• **gem**  
  Ruby

• ...