Flask

Flask: a "lightweight" framework for serving web applications.

Flask uses a simple dispatch mechanism for launching functions based on the URL

hello_flask.py

```
from flask import Flask
app = Flask(__name__)      # a Flask object

@app.route('/')            # called when visiting web URL 127.0.0.1:5000/
def hello_world():
    return 'Hello World!<BR><BR>Say <A HREF="/bye">Bye!</A>'

@app.route('/bye')         # called when visiting web URL 127.0.0.1:5000/bye
def goodbye():
    return 'Get thee to a nunnery!<BR><BR>Say <A HREF="/">Hi!</A>'

if __name__ == '__main__':
    app.run(debug=True, port=5000)    # app starts serving in debug mode on port 5000
```

The functions preceded by **@app.route()** decorators may be called *event functions*; these are called in response to an event, namely the calling of a URL by the user.

In the simplest application, returning a **string** from an event function causes Flask to return this string to the browser. (In most cases, the browser will interpret a string as HTML.)

Note the **<A HREF>** tags that call this app with one or the other page's URL: this app allows you to click a link that flips back and forth between pages.

Flask is among the simplest libraries for writing and serving web applications, and is perfectly suited to a wide range of domains. By contrast, Django is often used for larger-scale websites, although its configuration requirements are more demanding and its learning curve is steeper.

Running a Flask app locally

Flask comes complete with its own self-contained app server. We can simply run the app and it begins serving locally. No internet connection is required.

```
$ python hello_flask.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
*** IN hello_world() ***
127.0.0.1 - - [13/Nov/2016 15:58:16] "GET / HTTP/1.1" 200 -
*** IN goodbye() ***
127.0.0.1 - - [13/Nov/2016 15:58:17] "GET /bye HTTP/1.1" 200 -
*** IN hello_world() ***
127.0.0.1 - - [13/Nov/2016 15:58:18] "GET / HTTP/1.1" 200 -
```

The Flask app prints out web server log messages showing the URL requested by each visitor. You can also print error messages directly from the application, and they will appear in the log (these were printed with *** strings, for visibility.)

Redirection and Event Flow

After an 'event' function is called, it may return a page, or redirect to another function.

Return another function call
Functions that aren't intended to return pages but to perform other actions (such as making database changes) can simply call other functions that represent the desired destination:

```
@app.route('store_record')
def store_record():

    record_data = get_new_data()   # just hypothetical functions
    write_db(record_data)          # that update the database

    return list_records()          # redirect flow to another function
```

Of course whatever is returned from list_records() (presumably, an HTML page) will be returned from here as well.

Redirecting to another program URL with flask.redirect() and flask.url_for()
At the end of a function we can call the flask app again through a page redirect -- that is, to have the app call itself with new parameters.

```
@app.route('validate_login')
def validate_login():

    if not login_is_valid():        # hypothetical validation function

        return flask.redirect(flask.url_for('login', msg='login invalid'))   # redirect to login/
```

redirect() issues a redirection to a specified URL; this can be **http://www.google.com** or any desired URL.

url_for() simply produces the URL that will call the flask app with **/login** in addition to parameter arguments (discussed shortly).

Reading args from URL or Form Input

Input from a page can come from a link URL, or from a form submission.

reading parameter values from a URL
URL parameters appear in the URL as might be generated by a link:

```
<A HREF="http://127.0.0.1:5000/dothis?fname=David&lname=Blaikie">click me</A>
```

The code for retrieving these values:

```
@app.route('/insert_row')
def insert_row():
    fname = flask.request.args.get('fname')             # like dict.get()
    lname = flask.request.values['lname']               # like dict access
    insert_db(fname, lname)                             # hypothetical
    return flask.redirect(flask.url_for('display_data')) # generate URL to go to event function 'display_data'
```

reading parameter values from a form submission
Input from forms also generates parameter output, however forms with METHOD="POST" will not post values to the URL -- in these cases we can use the **form.args.get()** method:

```
@app.route('/insert_row')
def insert_row():
    fname = flask.form.args.get('fname')
    lname = flask.form.args.get('lname')
    insert_db(fname, lname)
    return flask.redirect(flask.url_for('display_data'))
```

Basic Templating with jinja2 Templates and flask.render_template()

Templates are HTML pages that contain dynamic data inserted by Flask. These can be variables, object attributes or method calls, or basic Python logic.

Template document: "index.html"

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Important Stuff</title>
  </head>
  <body>

  <h1>Important Stuff</h1>

  Today's magic number is {{ number }}<br><br>

  Today's insistent word is {{ word.upper() }}<br><br>

  Today's important topics are:<br>
  {% for item in mylist %}
      {{ item }}<br>
  {% endfor %}
  <br><br>

  {% if reliability_warning %}
      WARNING:  this information is not reliable
  {% endif %}

  </body>
</html>
```

Flask code

```
    return render_template('test.html', number=1035,
                                        word='somnolent',
                                        mylist=['children', 'animals', 'bacteria'],
                                        reliability_warning=True)
```

{{ variable }} can used for variable insertions, as well as instance attributes, method and function calls

{% for this in that %} can be used for 'if' tests, looping with 'for' and other basic control flow

Base templates

Many times we want to apply the same HTML formatting to a group of templates -- for example the <head> tag, which my include css formatting, javascript, etc. We can do this with base templates:

```
{% extends "base.html" %}         # 'base.html' can contain HTML from another template
    <h1>Special Stuff</h1>
    Here is some special stuff from the world of news.
```

The base template "surrounds" any template that imports it, inserting the importing template at the **{% block body%}** tag:

```
<html>
  <head>
  </head>
  <body>
    <div class="container">
      {% block body %}
         <H1>This is the base template default body.</H1>
      {% endblock %}
    </div>
  </body>
</html>
```

There are many other features (http://jinja.pocoo.org/docs/dev/templates/) of Jinja2 as well as ways to control the API, although I have found the above features to be adequate for my purposes.

Sessions

Sessions (usually supported by cookies) allow Flask to identify a user between requests (which are by nature "anonymous").

When a session is set, a cookie with a specific ID is passed from the server to the browser, which then returns the cookie on the next visit to the server. In this way the browser is constantly re-identifying itself through the ID on the cookie. This is how most websites keep track of a user's visits.

```
import flask
app = flask.Flask(__name__)

app.secret_key = 'A0Zr98j/3yX RXHH!jmN]LWX/,?RT'  # secret key

@app.route('/index')
def hello_world():

    # see if the 'login' link was clicked:  set a session ID
    user_id = flask.request.args.get('login')
    if user_id:
        flask.session['user_id'] = user_id
        is_session = True

    # else see if the 'logout' link was clicked:  clear the session
    elif flask.request.args.get('logout'):
        flask.session.clear()

    # else see if there is already a session cookie being passed:  retrieve the ID
    else:
        # see if a session cookie is already active between requests
        user_id = flask.session.get('user_id')

    tell the template whether we're logged in (user_id is a numeric ID, or None)
    return flask.render_template('session_test.html', is_session=user_id)


if __name__ == '__main__':
    app.run(debug=True, port=5001)    # app starts serving in debug mode on port 5001
```

The template below works with the above code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Session Test</title>
  </head>
  <body>

  <h1>Session Test</h1>

  {% if is_session %}
  <font color="green">Logged In</font>
  {% else %}
  <font color="red">Logged Out</font>
  {% endif %}

  <br><br>

  <a href="index?login=True">Log In</a><br>
  <a href="index?logout=True">Log Out</a><br>

  </body>
</html>
```

Config Values

Configuration values are set to control how Flask works as well as to be set and referenced by an individual application.

Flask sets a number of variables for its own behavior, among them **DEBUG=True** to display errors to the browser, and **SECRET_KEY='!jmNZ3yX RXWX/r]LA098j/,?RTHH'** to set a session cookie's secret key.

A list of Flask default configuration values is here. (http://flask.pocoo.org/docs/0.11/config/)

Retrieving config values

```
value = app.config['SERVER_NAME']
```

Setting config values individually

```
app.config['DEBUG'] = True
```

Setting config values from a file

```
app.config.from_pyfile('flaskapp.cfg')
```

Such a file need only contain python code that sets uppercased constants -- these will be added to the config.

Setting config values from a configuration Object

Similarly, the class variables defined within a custom class can be read and applied to the config with **app.config.from_object()**. Note in the example below that we can use inheritance to distribute configs among several classes, which can aid in organization and/or selection:

In a file called **configmodule.py**:

```
class Config(object):
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite://:memory:'

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
    DEBUG = True

class TestingConfig(Config):
    TESTING = True
```

In the flask script:

```
app.config.from_object('configmodule.ProductionConfig')
```

Environment Variables

Environment Variables are system-wide values that are set by the operating system and apply to all applications. They can also be set by individual applications.

The OpenShift web container sets a number of environment variables, among them **OPENSHIFT_LOG_DIR** for log files and **OPENSHIFT_DATA_DIR** for data files.

A list of Openshift environment variables can be found here. (https://developers.openshift.com/managing-your-applications/environment-variables.html)

Flask and security

An important caveat regarding web security: Flask is not considered to be a secure approach to handling sensitive data.

...at least, that was the opinion of a former student, a web programmer who worked for Bank of America, about a year ago -- they evaluated Flask and decided that it was not reliable and could have security vulnerabilities. His team decided to use **CGI** -- the baseline protocol for handling web requests.

Any framework is likely to have vulnerabilities -- only careful research and/or advice of a professional can ensure reliable privacy.

However for most applications security is not a concern -- you will simply want to avoid storing sensitive data on a server without considering security.