Webapps

Web applications

- Sooner or later, you'll want to share your app with lots of people…

- A "**webapp**" is what you want. If you develop your program as a *Web-based application* (or *webapp*, for short), your program is:
  1. Available to everyone who can get to your website
  2. In one place on your web server
  3. Easy to upate as new functionality is needed

- But...how do webapps actually work?

No matter what you do on the Web, it's all about *requests* and *responses*. A **web request** is sent from a web browser to a web server as the result of some user interaction. On the web server, a **web response** (or *reply*) is formulated and *sent back* to the web browser. The entire process can be summarized in five steps.

1. Your user enters a web address, selects a hyperlink, or clicks a button in her chosen web browser.
2. The web browser converts the user's action into a web request and sends it to a server over the Internet.
3. The web server receives the web request and has to decide what to do next.
4. The web server processes the web request, creating a web response, which is sent back over the Internet to the waiting web browser.
5. The web browser receives the web response and displays it on your user's screen.

Design

- Remember the exercise with the data series and the "Athletes": our last implementation, with classes, was *Class3.py*

  `http://bit.do/Class3-py`

- Now you want a friendly home page, from which every Athlete can read her times, nicely formatted for printing.

- So at minimum you need:
  - An home page with a friendly graphic and with a link to the webapp
  - A page that displays a list of all the series available. If you click on a series radio button and print the "select" button you can see the data
  - A third web page that displays the selected Athlete data, with links back to the other two pages.

MVC

- Now that you have an idea of the pages your webapp needs to provide, your next question should be: *what's the best way to build this thing?*
- Great webapps, and great web frameworks, conform to the *Model-View-Controller* pattern, which helps you segment your webapp's code into easily manageable functional chunks (or *components*):
  1. **The Model**: The code to store (and sometimes process) your webapp's data
  2. **The View:** The code to format and display your webapp's user interface(s)
  3. **The Controller**: The code to glue your webapp together and provide its business logic

- By following the MVC pattern, you build your webapp in such as way as to enable your webapp to grow as new requirements dictate. You also open up the possibility of splitting the workload among a number of people, one for each component.

Model your data

- Your web server needs to store a single copy of your data, which in this case is our timing values in text files.
- When your webapp **starts**, the data in the text files needs to be converted to AthleteList object instances, stored within a dictionary (indexed by athlete name), and then saved as a pickle file. Let's put this functionality in a new function called `put_to_store()`.
- While your webapp **runs**, the data in the pickle needs to be available to your webapp as a dictionary. Let's put this functionality in another new function called `get_from_store()`.

http://bit.do/AthleteModel-py

```
import pickle
def sanitize(time_string):
class AthleteList(list):
def get_run_data(filename):

def put_to_store(files_list):
    all_athletes = {}
    for each_file in files_list:
        ath = get_run_data(each_file)
        all_athletes[ath.name] = ath
    try:
        with open('athletes.pickle', 'wb') as athf:
            pickle.dump(all_athletes, athf)
    except IOError as ioerr:
        print 'File error (put_and_store): ' + str(ioerr)
    return all_athletes

def get_from_store():
    all_athletes = {}
    try:
        with open('athletes.pickle', 'rb') as athf:
            all_athletes = pickle.load(athf)
    except IOError as ioerr:
        print 'File error (get_from_store): ' + str(ioerr)
    return all_athletes
```

Same as in Class3.py

Take each file, turn it into an AthleteList object instance, and add the athlete's data to the dictionary.

Each athlete's name is used as the "key" in the dictionary. The "value" is the AthleteList object instance.

Save the entire dictionary of AthleteLists to a pickle

Simply read the entire pickle into the dictionary.

If you have downloaded all the files needed (AthleteModel.py + *name*2.txt text files), you can now test the functions. Remember that a module can be imported if the files is in the working directory; if instead is in the HOME/my/modules dir you can run:

```
module_dir=os.path.join(os.environ['HOME'],'my','modules')
sys.path.insert(0,module_dir)
```

```
>>> import AthleteModel as am
>>> the_files = ['anna2.txt', 'giulia2.txt', 'rosa2.txt', 'sonia2.txt']
>>> data = am.put_to_store(the_files)
>>> data
{'Giulia Sagramola': ['2.59', '2.11', '2:11', '2:23', '3-10', '2-23', '3:10',
'3.21', '3-21', '3.01', '3.02', '2:59'], 'Sonia Gandhi': ['2:58', '2.58', '2:39',
'2-25', '2-55', '2:54', '2.18', '2:55', '2:55', '2:22', '2-21', '2.22'], 'Anna
Magnani': ['2-34', '3:21', '2.34', '2.45', '3.01', '2:01', '2:01', '3:10', '2-22',
'2-01', '2.01', '2:16'], 'Rosa Aulente': ['2:22', '3.01', '3:01', '3.02', '3:02',
'3.02', '3:22', '2.49', '2:38', '2:40', '2.22', '2-31']}
>>> type(data)
<type 'dict'>
```

So here's all of the AthleteLists. At this point, the `athletes.pickle` file should appear in the same folder as your code and text files.

```
>>> for each_athlete in data:
        print(data[each_athlete].name + ' ' + data[each_athlete].dob)
Giulia Sagramola 2002-8-17
Sonia Gandhi 2002-6-17
Anna Magnani 2002-3-14
Rosa Aulente 2002-2-24
```

Now try to use the `get_from_store()` function to load the pickled data into another dictionary, then confirm that the results are as expected by repeating the code to display each athlete's name and date of birth:

```
>>> data_copy = am.get_from_store()
>>> for each_athlete in data_copy:
        print(data_copy[each_athlete].name + ' ' + data_copy[each_athlete].dob)
Giulia Sagramola 2002-8-17
Sonia Gandhi 2002-6-17
Anna Magnani 2002-3-14
Rosa Aulente 2002-2-24
>>> data is data_copy
False
```

With your model code written and working, it's time to look at your view code, which creates your webapp's user interface (UI).

YATE: Yet Another Template Engine

- On the Web, UIs are created with HTML, the Web's markup technology.
- Here we will use a small module that might help you generate HTML. It's a little rough, but it works. It is provided by the authors of our reference book.
- Let's get to know the yate code before proceeding. For each chunk of code presented we'll see a brief explanation.

http://bit.do/yate-py

http://v.gd/uxacib (header.html)

http://v.gd/ezisul (footer.html)

```
from string import Template

def start_response(resp="text/html"):
    return('Content-type: ' + resp + '\n\n')

def include_header(the_title):
    with open('header.html') as headf:
        head_text = headf.read()
    header = Template(head_text)
    return(header.substitute(title=the_title))

def include_footer(the_links):
    with open('footer.html') as footf:
        foot_text = footf.read()
    link_string = ''
    for key in the_links:
        link_string += '<a href="' + the_links[key] + '">' + key +
'</a>    '
    footer = Template(foot_text)
    return(footer.substitute(links=link_string))

def start_form(the_url, form_type="POST"):
    return('<form action="' + the_url + '" method="' + form_type + '">')

def end_form(submit_msg="Submit"):
    return('<p></p><input type=submit value="' + submit_msg + '"></form>')
```

Import the "Template" class from the standard library's "string" module. This allows for simple string-substitution templates.

This function takes a single (optional) string as its argument and uses it to create a CGI "Content-type:" line, with "text/html" as the **default**.

This function takes a single string as its argument and uses it as the title for the start of a HTML page. The page itself is stored within a separate file in "header.html", and the title is substituted in as needed

This one uses its single string as its argument to create the end of a HTML page. The page itself is stored within a separate file in "footer.html", and the argument is used to dynamically create a set of HTML link tags. **Based on how they are used, it looks like the argument needs to be a dictionary.**

This function returns the HTML for the start of a form and lets the caller specify the URL to send the form's data to, as well as the method to use.

This function returns the HTML markup which terminates the form while allowing the caller to customize the text of the form's "submit" button.

```
def radio_button(rb_name, rb_value):
    return('<input type="radio" name="' + rb_name +
                                '" value="' + rb_value + '"> ' + rb_value + '<br />')
def u_list(items):
    u_string = '<ul>'
    for item in items:
        u_string += '<li>' + item + '</li>'
    u_string += '</ul>'
    return(u_string)

def header(header_text, header_level=2):
    return('<h' + str(header_level) + '>' + header_text +
           '</h' + str(header_level) + '>')

def para(para_text):
    return('<p>' + para_text + '</p>')
```

Given a radio-button name and value, create a HTML radio button (which is typically included within a HTML form). Note: both arguments are required.

Given a list of items, this function turns the list into a HTML unnumbered list. A simple “for” loop does all the work, adding a LI to the UL element with each iteration.

Create and return a HTML header tag (H1, H2, H2, and so on) with level 2
as the default. The "header_text" argument is required.

Enclose a paragraph of text (a string) in HTML paragraph tags.

You could also include the HTML that you need right in the code and generate it with `print` as needed , but it’s not as flexible as the approach shown here. Using a collection of `print` statements to generate HTML works, but it turns your code into an unholy mess, breaking the MVC paradigm.

Let's check interactively the View module functions

```
>>> import yate as y
>>> y.start_response()
'Content-type: text/html\n\n'
```

The default CGI response header.

```
>>> y.include_header("Welcome to my home on the web!")
'<html>\n<head>\n<title>Welcome to my home on the web!</title>\n<link
type="text/css" rel="stylesheet" href="/coach.css" />\n</head>\n<body>\n<h1>Welcome
to my home on the web!</h1>\n'
```

HTML output. Note the inclusion of a link to a CSS file

```
>>> y.include_footer({'Home': '/index.html', 'Select': '/cgi-bin/select.py'})
'<p>\n<a href="/index.html">Home</a>    <a href="/cgi-
bin/select.py">Select</a>    \n</p>\n</body>\n</html>\n'


>>> y.include_footer({})
'<p>\n\n</p>\n</body>\n</html>\n'


>>> y.start_form("/cgi-bin/process-athlete.py")
'<form action="/cgi-bin/process-athlete.py" method="POST">'

>>> y.header("This is a sub-sub-sub-sub heading", 5)
'<h5>This is a sub-sub-sub-sub heading</h5>'
```

Control your code

Your **model** code is ready, and you have a good idea of how the `yate` module can help you with your **view** code. It's time to glue it all together with some **controller** code.

It is recommended to organize the webapp's directory structure as follows:

- **webapp** As well as containing the subfolders, this folder contains your webapps "index.html" file, your "favicon.ico" icon, style sheets, and anything else that doesn't fit neatly into one of the subfolders
  - **cgi-bin** Any code that you write for your webapp needs to reside in a specially named folder called "cgi-bin".
  - **data** Let's keep the data files in a separate folder by putting all of the TXT files in here
  - **images** If your webapp has any images files (JPGs, GIFs, PNGs, and so on), pop them into their own folder to help keep things organized.
  - **templates** The templates that came with the "yate.py" can go in here

Download and expand the zipped folder http://bit.do/webapp-zip

CGI

- The Common Gateway Interface (CGI) is an Internet standard that allows for a web server to run a **server-side program**, known as a *CGI script*.
- Typically, CGI scripts are placed inside a special folder called cgi-bin, so that the web server knows where to find them. On some operating systems (most notably UNIX-styled systems), CGI scripts must be set to *executable* before the web server can execute them when responding to a web request.
- Practically every web server on the planet supports CGI, but using one of these tools here is *overkill*

Simple Server

Python comes with its very own web server. Check the contents of the webapp.zip download: it comes with a CGI enabled web server (`simple_httpd.py`).

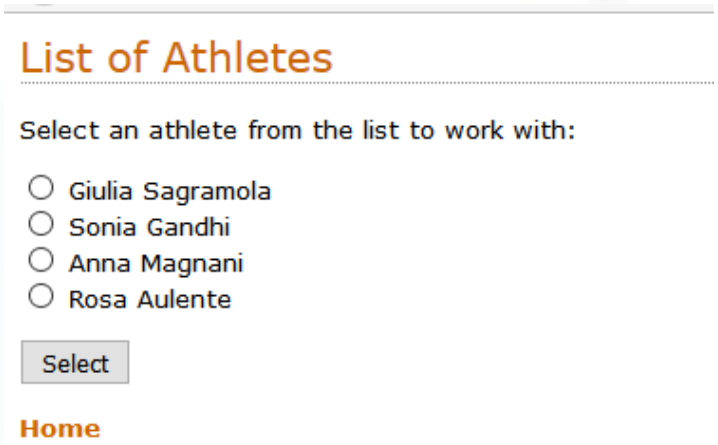
```
from BaseHTTPServer import HTTPServer
from CGIHTTPServer import CGIHTTPRequestHandler
port = 8080

httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
print("Starting simple_httpd on port: " + str(httpd.server_port))
httpd.serve_forever()
```

3.x

```
from http.server import HTTPServer, CGIHTTPRequestHandler

port = 8080

httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
print("Starting simple_httpd on port: " + str(httpd.server_port))
httpd.serve_forever()
```

Control

- Let's create a program called `generate_list.py` which, when executed by the web server, dynamically generates a HTML web page like this:



- This will be the second web server page, following the presentation page served by `index.html`
- When your user selects an athlete by clicking on her radio button and clicking Select, a *new* web request is sent to the web server. This new web request contains data about which radio button was pressed, *as well as the name of a CGI script to send the form's data to.*
- Recall that all of your CGI scripts need to reside in the cgi-bin folder on your web server.

http://bit.do/generate_list-py

```
import athletemodel
import yate
import glob

data_files = glob.glob("data/*.txt")
athletes = athletemodel.put_to_store(data_files)
```

Use your `put_to_store()` function to create a dictionary of athletes from the list of data files.

Start generating the form, providing the name of the serverside program to link to.

```
print yate.start_response()
print yate.include_header("List of Athletes")
print yate.start_form("generate_timing_data.py")
print yate.para("Select an athlete from the list to work with:")
```

Generate a radiobutton for each of your athletes.

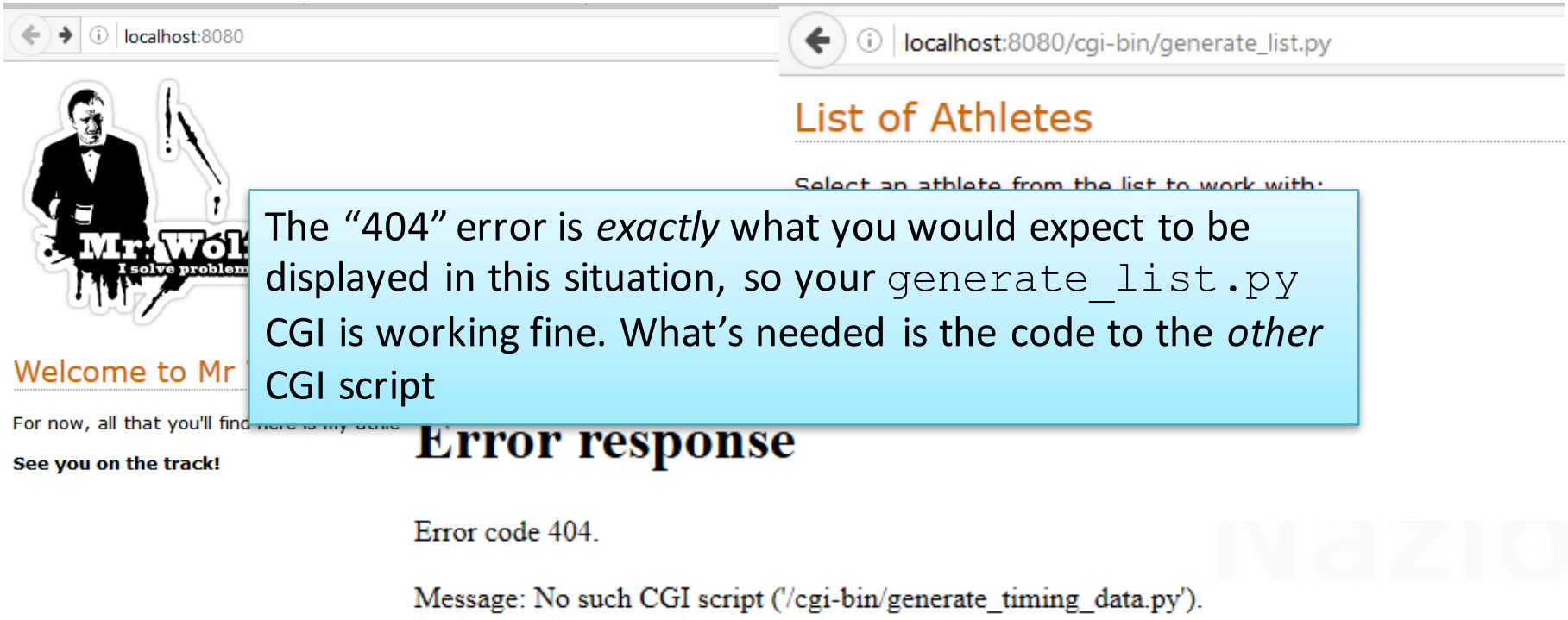
```
for each_athlete in athletes:
    print yate.radio_button("which_athlete", athletes[each_athlete].name)
print yate.end_form("Select"))
```

Let's add a link to the bottom of the generated HTML page that takes your user home

```
print yate.include_footer({"Home": "/index.html"})
```

First run

Open the script `simple_httpd.py` with IDLE (Ctrl+O) ad run it (F5).
Now your web server is running, and you can contact it:
http://localhost:8080



The "404" error is *exactly* what you would expect to be displayed in this situation, so your `generate_list.py` CGI is working fine. What's needed is the code to the *other* CGI script

```
127.0.0.1 - - [22/Apr/2016 22:26:46] code 404, message No such CGI script
('/cgi-bin/generate_timing_data.py')
127.0.0.1 - - [22/Apr/2016 22:26:46] "POST /cgi-bin/generate_timing_data.py
HTTP/1.1" 404 -
```

CGI form data

- Let's take a moment to recall what is required from the `generate_timing_data.py` CGI script: you need to generate a new HTML page that contains the top three times for the **selected athlete**

- When you click on a radio-button and then press the Select button, a new web request is sent to the server. The web request identifies the CGI script to execute (in this case, that's `generate_timing_data.py`), **together with the form's data**. The web server arranges to send the form's data to your CGI script as its input. Within your code, you can access the form data using Python's cgi module, which is part of the standard library.

```
import cgi
form_data = cgi.FieldStorage()
athlete_name = form_data['which_athlete'].value
```

Access a named piece of data from the form's data

http://bit.do/generate_timing_data-py

```
import cgi
import athletemodel
import yate

athletes = athletemodel.get_from_store()


form_data = cgi.FieldStorage()
athlete_name = form_data['which_athlete'].value

print yate.start_response()
print yate.include_header("Coach Kelly's Timing Data")
print yate.header("Athlete: " + athlete_name + ", DOB: " +
athletes[athlete_name].dob + ".")

print yate.para("The top times for this athlete are:")
print yate.u_list(athletes[athlete_name].top3())
print yate.include_footer({"Home": "/index.html", "Select another athlete":
"generate_list.py"})
```

Get the data from the model

Which athlete's data are you working with?

Page generation

Type Error!

- If you run the web site, and go to the times page, you will see that the first section of it it's ok, but you cannot see the times.
- It's not clear *on the web browser screen* that anything has gone wrong, but if you check the web server logging you will see something similar to:

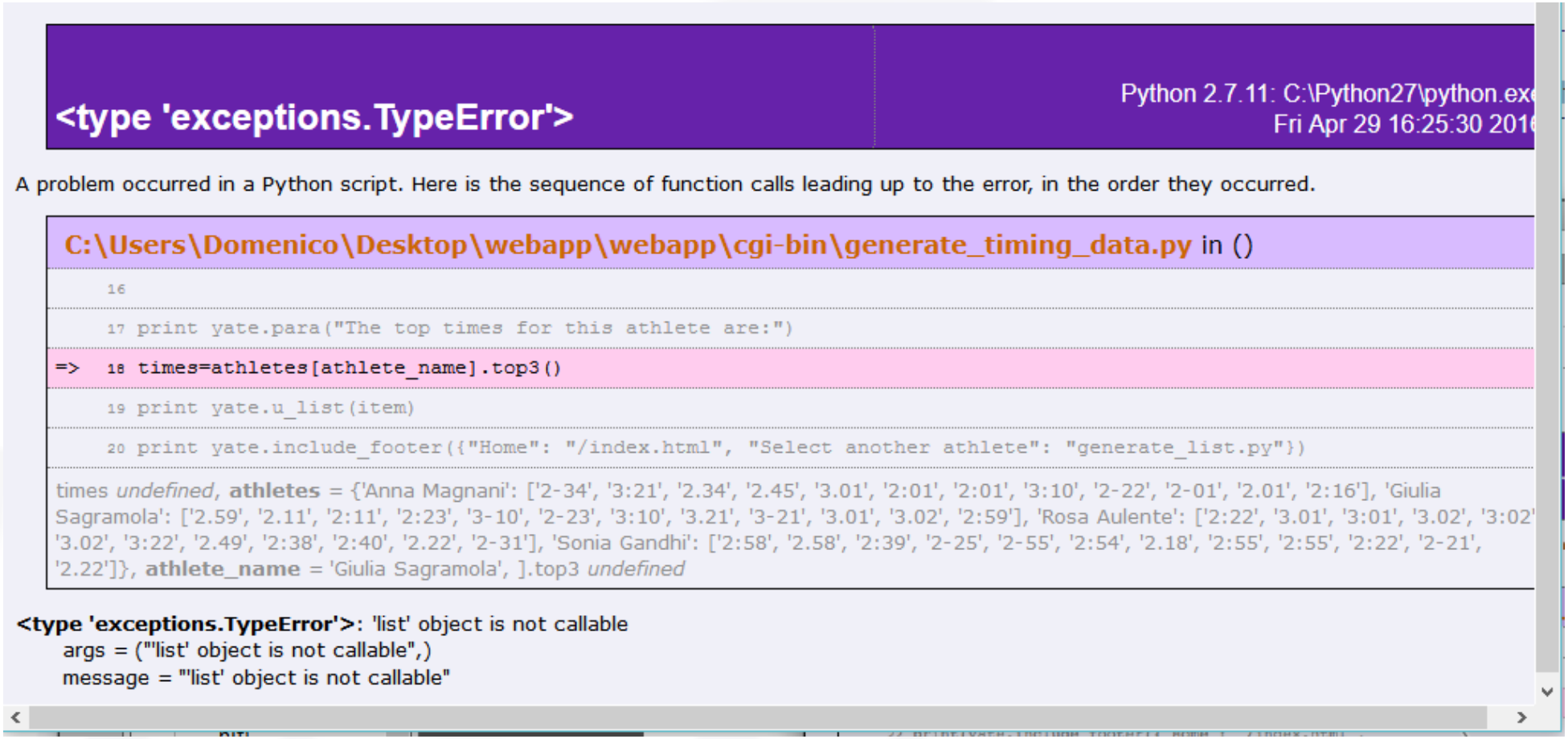
```
127.0.0.1 - - [28/Apr/2016 13:17:05] "POST /cgi-bin/generate_timing_data.py
HTTP/1.1" 200 -
127.0.0.1 - - [28/Apr/2016 13:17:05] command: C:\Python27\python.exe -u
C:\Users\Domenico\Desktop\webapp\webapp\cgi-bin\generate_timing_data.py ""
127.0.0.1 - - [28/Apr/2016 13:17:05] Traceback (most recent call last):

  File "C:\Users\Domenico\Desktop\webapp\webapp\cgi-
bin\generate_timing_data.py", line 16, in <module>

    times=athletes[athlete_name].top3()

TypeError: 'list' object is not callable


127.0.0.1 - - [28/Apr/2016 13:17:05] CGI script exit status 0x1
```

Enable CGI tracking

- Python's standard library comes with a CGI tracking module (called `cgitb`) that, when enabled, arranges for detailed error messages to appear in your web browser. These messages can help you work out where your CGI has gone wrong. When you've fixed the error and your CGI is working well, simply switch off CGI tracking.

```
import cgitb
cgitb.enable()
```

- This lines must be added to the `generate_timing_data.py` CGI script



The CGI tracking output indicates an error with the use of the top3() method from the AthleteList code.

A quick review of the code to the AthleteList class uncovers the source of the error: the top3() method has been redesignated as a **class property**

Decorators

- Decorators are a syntactic convenience, that allows a Python source file to say what it is going to do with the result of a function or a class statement **before** rather than after the statement. The reader knows, before the possibly quite long definition of the function, that the decorator function will be applied to it.
- The decorator syntax uses the @ character. For function statements the following are equivalent:

```
# State, before defining f, that a_decorator will be applied to it.
@a_decorator
def f(...):
```

```
def f(...):
    ...

# After defining f, apply a_decorator to it.
f = a_decorator(f)
```

AthleteList

- **@staticmethod**
  With staticmethods, neither self (the object instance) nor cls (the class) is implicitly passed as the first argument. They behave like plain functions except that you can call them from an instance or the class. Staticmethods are used to group functions which have some logical connection with a class to the class. Static methods are an organization/stylistic feature.

- **@property**
  This decorator allows you to access the data returned as if it were a class attribute.

@property

- So you must treat the top3() method as if it was another class attribute, and call it like this, without parentheses:

```
print(yate.u_list(athletes[athlete_name].top3))
```

- If you modify the code of `generate_timing_data.py` now finally the web site works as expected.

GAE

Google App Engine

- When your webapp goes from a handful of hits a day to thousands, possibly ten of thousands, or even more, will your web server handle the load? How will you know? What will it cost? Who will pay? Can your data model scale to millions upon millions of data items without slowing to a crawl?

- Getting a webapp up and running is easy with Python and now, thanks to Google App Engine, scaling a Python webapp is achievable, too.

- Google App Engine (GAE) is a set of technologies that lets you host your webapp on Google's cloud computing infrastructure.
  You can then avoid to invest in a large, state-of-the-art web server that can be hosted in your central office (with the setup and the broadband link required), or expensive web hosting solution

Google App Engine

- GAE constantly monitors your running webapp and, based on your webapp's current activity, adjusts the resources needed to serve up your webapp's pages.
- When things are busy, GAE increases the resources available to your webapp, and when things are quiet, GAE reduces the resources until such time as extra activity warrants increasing them again.
- On top of this, GAE provides access to Google's *BigTable* technology: a set of database technologies. Google also backs up your webapp's data on a regular basis, replicates your webapp over multiple, geographically dispersed web servers, and keeps App Engine running smoothly 24/7.
- And the *best part*? GAE can be programmed with Python.
- And the *even better part*? You can start running your webapp on GAE *for free*.

	FREE *LIMIT PER DAY*	PRICE *ABOVE FREE LIMIT*
Instances	28 instance hours	$0.05 / instance / hour
Cloud Datastore *(NoSQL)*	• 50k read/write/small • 1 GB storage	• $0.06 / 100k read or write ops • Small operations free* • $0.18 / GB / month
Network Traffic *(Outgoing)*	1 GB	$0.12 / GB
Network Traffic *(Incoming)*	1 GB	FREE
Cloud Storage	5 GB	$0.026 / GB / month
Memcache	• Free Usage of Shared Pool • No free quota for Dedicated Pool	• Free Usage of Shared Pool • Dedicated Pool: $0.06 / GB / hour
Search	• 1000 basic operations • 0.01 GB indexing documents • 0.25 GB document storage • 100 searches	• 0.50 / 10k searches • 2.00 / GB indexing documents • 0.18 / GB / month Storage
Email API	100 recipients	Contact Sales
Logs API	100 MB	$0.12 per GB
Task Queue	5 GB	$0.026 / GB / month
Logs Storage	1 GB	$0.026 / GB / month
SSL Virtual IPs	-	$39 / virtual IP / month
Bundled Services	Cron, Image Manipulation, SNI SSL Certificates, Socket API, Task Queue API, URLFetch, Users API	

Install GAE SDK

- When your webapp is ready for deployment, you'll upload it to the Google cloud and run it from there. However, during development, you can run a test version of your webapp locally on your computer. All you need is a copy of the GAE SDK, which is available from here:

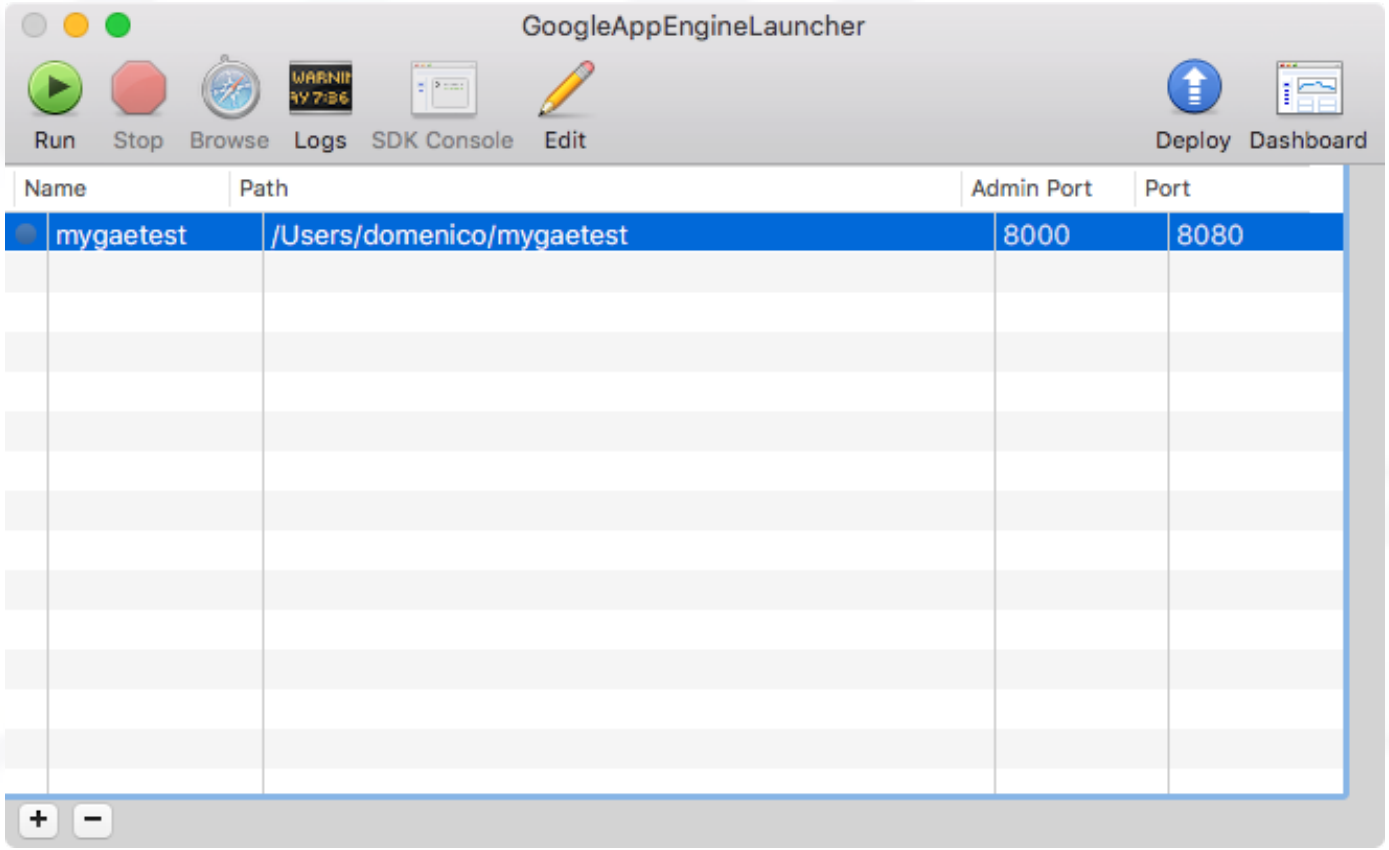
  http://v.gd/yimera

- After installation, Windows and Mac OS X users will find a graphical front end added to their system. On Linux, a new folder called "google_appengine" is created after a successful install.

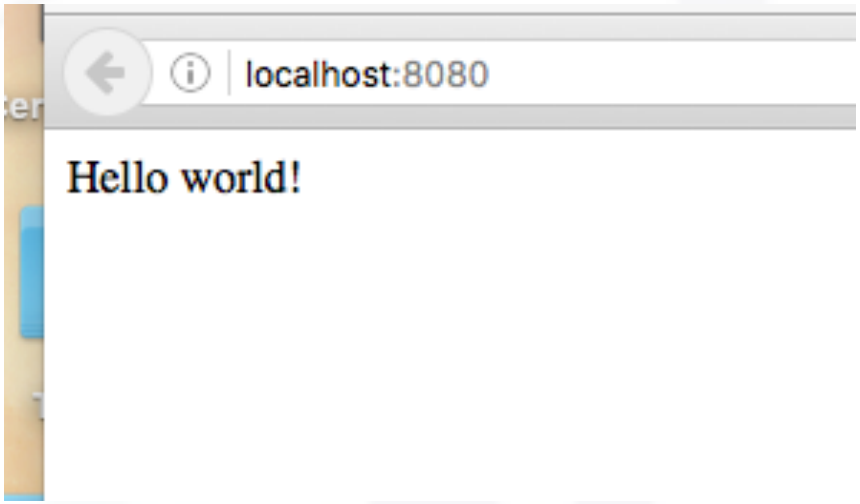
- GAE uses Python 2.7. If the installer does not find it, give it its path.

Make sure App Engine is working

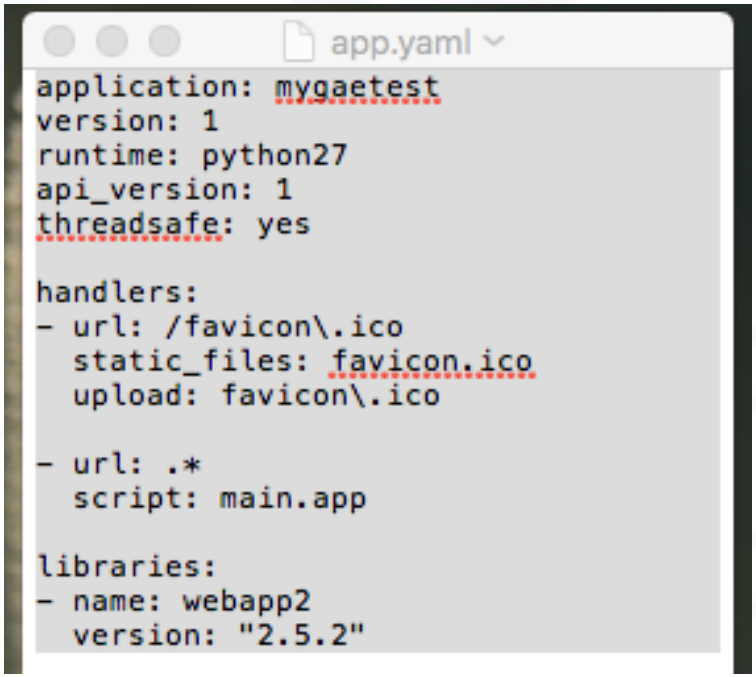
- To build a GAE-compatible webapp, you need three things: a **folder** to hold your webapp's files, some **code** to execute, and a **configuration file**.
- To test your setup, create a folder called *mygaetest*. (file->New Application).



The basic application files are automatically created. Now you can *run* the test: click *browse and* authorize the connections on the port 8080. You must have something like this:



If you click on *edit* you will see the configuration file of your application



GAE and MVC

- GAE-enabled webapp uses a back-end data storage facility that's known as the **datastore**. This is based on Google's *BigTable* technology, which provides a "NoSQL" API to your data, as well as a SQL-like API using Google's Query Language (GQL).
- GAE uses the **templating system** from the Django Project, which is one of Python's leading web framework technologies. In addition to templates, GAE includes Django's forms-building technology.
- And, of course, any controller **code** is written in Python and can use the CGI or WSGI standards.
- So you must define a **model** for your data, create some **templates** for your view, and then control it all with **code**.

Model your data with App Engine

- App Engine refers to data items stored within its datastore as *properties*, which are defined within your model code.
- Think of properties as a way to define the name and types of data within your database schema: each property is like the column type associated piece of data stored in a row, which App Engine refers to as an *entity*.
- When you think “row,” GAE thinks “entity.” And when your think “column,” GAE thinks “property.”

Data types

- As with traditional SQL-based databases, your GAE datastore properties are of a specific, predeclared type, for istance:
  - **db.StringProperty**: a string of up to 500 characters
  - **db.Blob**: a byte string (binary data)
  - **db.DateProperty**: a date
  - **db.TimeProperty**: a time,
  - **db.IntegerProperty**: a 64-bit integer
  - **db.UserProperty**: a Google account

- At this link

  `http://v.gd/bavipe`

  you will find the full documentation. In particular all the supported types are here:

  `http://v.gd/fihuye`

ID	NAME	DOB
1	Sarah Connor	31-12-2000
2	Sonia Ghandi	12-03-1998
3	Anna Magnani	03-03-2003
4	Rosa Aulente	01-01-1990
5	Giulia Sagramola	03-10-2010

This is a db.IntegerProperty

This is a db.StringProperty

This is a db.DateProperty

Define the data

- We want to draw a web form, that allows the user to save a bunch of data, say a problem report of a data farm.
- Our data will be Name and Email of who fills the form, Date and Time of the problem, the name of the Server on which the problem had been detected, the name of the faulted Service. We create a new application folder `farmproblem`
- The first task is to create in the app folder a file, say `problemDB.py`, with the definition of a class that inherits from the GAE `db.Model` class, and that assign to each needed property a name.

```
from google.appengine.ext import db
class farmProblem(db.Model):
    name = db.StringProperty()
    email = db.StringProperty()
    date = db.DateProperty()
    time = db.TimeProperty()
    server = db.StringProperty()
    service = db.StringProperty()
```

Let's take a view

- GAE not only lets you *define* the schema for your data, but it also *creates* the entities in the datastore. The first time you go to **put** your data in the datastore, GAE springs to life and makes room for your data.

- But first you have to get some data from your webapp's user...and to do that, you need a *view*. And views are easy when you use *templates*.

- The templating technology built into GAE is based on technology from the Django Project. Django's templating system is more sophisticated than the simple string-based templates used in the previous slides. Like your templates, Django's templates can substitute data into HTML, but they can also execute *conditional* and *looping* code.

- Here are four templates you'll need for your webapp. As you can see, rather that using the `$name` syntax for variable substitution in the template, Django uses the `{{name}}` syntax. Put them in a `templates` subdirectory.

header.html

```
<html>
<head>
<title>{{ title }}</title> </head>
<body>
<h1>{{ title }}</h1>
```

footer.html

```
<p> {{ links }} </p> </body> </html>
```

form_start.html

```
<form method="POST" action="/"> <table>
```

form_end.html

```
<tr><th> </th><td><input type="submit" value="{{sub_title}}"></td></tr>
</table>
</form>
```

Using templates

- To use a template, you must import the `template` module from `google.appengine.ext.webapp` and call the `template.render()` function.
  It is useful to assign the output from `template.render()` to a variable, which is called ***`html`*** in this code snippet

```
from google.appengine.ext.webapp import template
html = template.render('templates/header.html', {'title': 'Report a Service
Problem in Farm'})
html = html + template.render('templates/form_start.html', {})

#FORM CREATION

html = html + template.render('templates/form_end.html', {'sub_title':
'Submit Service Problem'})
html = html + template.render('templates/footer.html', {'links': ''})
```

- The `render()` function always expects two arguments. If you don't need the second one, be sure to pass an empty dictionary.

Django's form validation framework

- Templates aren't the only things that App Engine "borrows" from Django. It also uses its form-generating technology known as the *Form Validation Framework*. Given a data model, GAE can use the framework to generate the HTML needed to display the form's fields within a HTML table.

- This model is used with Django's framework to generate the HTML markup needed to render the data-entry form. All you need to do is inherit from a GAE-included class called `djangoforms.ModelForm`. The framework generates the HTML you need

Controlling your App Engine webapp

- In the application folder you must create the `problem.py` file that must contain the controller code.
- Create also, in the application folder, a empty file named `settings.py`, that must be present in order to let the Django import correctly work.
- In the next slide we'll comment the controller code. Once it is written in the app folder you can Add an existing app in the GAE Launcher, play the app and browse it. You will see this page:

`http://bit.do/farmproblem-zip`

Import App Engine's "webapp" class.

Import a utility that runs your webapp (WSGI)

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db
from google.appengine.ext.webapp import template
import os
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
from google.appengine.ext.db import djangoforms

import problemDB

class ProblemForm(djangoforms.ModelForm):
    class Meta:
        model = problemDB.farmProblem

class ProblemInputPage(webapp.RequestHandler):
    def get(self):
        html = template.render('templates/head
Problem'})
        html = html + template.render('templates/form_start.html', {})

        html = html + str(ProblemForm())

        html = html + template.render('templates/form_end.html', {'sub_title':
'Submit Sighting'})
        html = html + template.render('templates/footer.html', {'links': ''})
        self.response.out.write(html)

app = webapp.WSGIApplication([('/.*', ProblemInputPage)], debug=True
```

Import db and template handlers

Needed to use Django

Your GAE data model code

Use your model to create a form that inherits from the "django.ModelForm" class.

The connected handler class is called "ProblemInputPage" and it provides a method called "get" which responds to a GET web request.

Include the generated form in the HTML response

Send a response back to the waiting web browser

Create an new "webapp" object for your application.

```
application: problem
version: 1
runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  script: problem.app

libraries:
- name: webapp2
  version: "2.5.2"

- name: django
  version: latest
```

The “application” line identifies your webapp and is the same name as your folder

The “version” line identifies the current version of your webapp (and usually starts at 1).

“runtime” tells GAE that your webapp is written in and will run on Python2.7

The “api_version” indicates the release of GAE you are targeting.

The “handlers” section of the configuration file is the top-level webapp routing mechanism.

The .* entry tells GAE to route all requests to your webapp to your problem app.

A library entry is equivalent to:
from google.appengine.dist import use_library
use_library('django', '0.96')

As any other HTML page, also this autogenerate form can be customized with the use of Cascading Style Sheets (CSS). You can expand the content of the following zip file in the subdirectory static of the application directory. The favico must go in the top level dir

http://bit.do/static-zip

Cosmetic touches

- To integrate the stylesheets into your webapp, add two link tags to your `header.html` template within your `templates` folder, in the `head` section. Here's what the tags need to look like:

```
<link type="text/css" rel="stylesheet" href="/static/hfwwg.css" />
<link type="text/css" rel="stylesheet" href="/static/styledform.css" />
```

- GAE is smart enough to *optimize* the delivery of static content—that is, content that does *not* need to be generated by code. Your CSS files are static and are in your static folder. All you need to do is tell GAE about them to enable optimization. Do this by adding the following lines to the handlers section of your `app.yaml` file, **BEFORE** the application handler:

```
- url: /static
  static_dir: static
```

Restrict input

- Providing a list of choices restricts what users can input. Instead of using HTML's INPUT tag for all of your form fields, you can use the SELECT/OPTION tag pairing to *restrict* what's accepted as valid data for any of the fields on your form.
- All you have to provide is the list of data items to use as an argument called `choices` when defining your property in your model code. You can also indicate when multiple lines of input are acceptable using the `multiline` argument to a property.
- Apply these changes to your **model** code in the **problemDB.py** file.

```
from google.appengine.ext import db

_SERVER = ['hpc0902', 'hpc0001', 'hpc2345']
_SERVICES = ['DNS', 'HTTP', 'HTTPS', 'SNMP']

class farmProblem(db.Model):
    name = db.StringProperty(multiline=True)
    email = db.StringProperty()
    date = db.DateProperty()
    time = db.TimeProperty()
    server = db.StringProperty(choices=_SERVER)
    service = db.StringProperty(choices=_SERVICES)
```

405 Method Not Allowed

- If you press submit, you will be greeted by this error. What the 405 status code actually tells you is that posted data arrived at your webapp intact, but that your webapp *does not* have any way of processing it. There's a method missing.

- In fact the only method currently defined in the controller `problem.py` is called `get()`. This method is invoked whenever a GET web request arrives at your webapp and, as you know, it displays your form.

- In order to process posted data, you need to define *another* method. Specifically, you need to add a new method called `post()` to your `ProblemInputPage` class.

Post()

- The `post()` method must gather the data from your web form, and put them in the GAE datastore. So in this function you have to :
  - **create** a new object from your data model
  - **get** the data from your HTML form
  - **assign** it to the object's attributes, and then use the `put()` method to
  - **save** your data in the datastore.

- It is also a good practice to generate a response page with a feedback.

```
def post(self):
    new_problem = problemDB.farmProblem()

    new_problem.name = self.request.get('name')
    new_problem.email = self.request.get('email')
    new_problem.date = self.request.get('date')
    new_problem.time = self.request.get('time')
    new_problem.server = self.request.get('server')
    new_problem.service = self.request.get('service')

    new_problem.put()

    html = template.render('templates/header.html', {'title': 'Thank you!'})
    html = html + "<p>Thank you for your report.</p>"
    html = html + template.render('templates/footer.html',
{'links': 'Enter <a href="/">another problem</a>.'})

    self.response.out.write(html)
```

Create a new "problem" object

For each of the data values received from the HTML form, assign them to the attributes of the newly created object

Store your populated object in the GAE datastore.

Generate a HTML response to say “thanks.” and send it to the browser

Now if you try to insert a record, you will have another problem:

```
BadValueError: Property date must be a date, but was u'12-01-
2016'
```

Date and time here are defined as `db.DateProperty()` and `db.TimeProperty()`, but there are may ways to enter a date and a time…

Date and Time

- If you are going to insist on asking your users to provide a properly formatted date and time, you'll need to do one of two things:

1. Specify in detail the **format** in which you expect the data.
2. **Convert** the entered data into a format with which you can work.

If you can use `db.StringProperty()` for dates and times

- For example, if you are too picky in requesting a date in a particular format, you'll slow down your user and might end up picking a date format that is foreign to them, resulting in confusion.

- If you try to convert *any* date or time entered into a common format that the datastore understands, you'll be biting off more than you can chew. As an example of the complexity that can occur, how do you know if your user entered a date in mm/dd/yyyy or dd/mm/yyyy format? (You don't.)

Developer console

- With a few problems entered, let's use App Engine's included **developer console** to confirm that the problems are in the datastore.
- Click on SDK console, Datastore Viewer, to see the entry generated by your form.
- AppEngine has assigned a Key and a ID to each of your entities; it stores all data entered in alphabetical order.
- You can use the console also to create a new entity, or to vies other datastore properties

Login

- The engineers at Google designed App Engine to deploy on Google's cloud infrastructure. As such, they decided to allow webapps running on GAE to access the *Google Accounts* system.
- By switching on **authorization**, you can require users of your webapp to log into their Google account *before* they see your webapp's pages. If a user tries to access your webapp and he isn't not logged in, GAE redirects to the Google Accounts login and registration page. Then, after a successful login, GAE returns the user to your waiting webapp.
- To switch on authorization, you have to make one small change to your app.yaml file:

```
- url: .*
  script: problem.app
  login: required
```

Deploy it!

- This is a two-step process: *register* and *upload*. To register your webapp on the Google cloud, click the **Dashboard** button on the GAE Launcher.
- The “Dashboard” button opens your web browser and takes you to the GAE “My Applications” page (after you sign in with your Google ID).
- From "Select a project" click on Create a Project: **the project-ID must match the application name in `app.yaml`**
- Now return on the GAE launcher and click deploy: if all goes well you will have in the log:

```
*** appcfg.py has finished with exit code 0***
```

Use it!

- Open your web browser and surf to a web address that starts with your webapp's name and ends in *.appspot.com*.
- When you first attempt to go to their webapp, App Engine redirects you to the Google login page. After a successful login, your form appears. Go ahead and enter some test data.
- Return to the

  `https://console.cloud.google.com`

  site to log into the console. The UI is a little different than the test console, but you can use the Datastore Viewer to confirm that your data has been stored correctly.

- https://problem-1309.appspot.com/

`http://bit.do/farmproblem2-zip`


That's all Folks!