EXERCISE 50

Your First Website

These final three exercises will be very hard and you should take your time with them. In this first one you'll build a simple web version of one of your games. Before you attempt this exercise you *must* have completed Exercise 46 successfully and have a working `pip` installed such that you can install packages and know how to make a skeleton project directory. If you don't remember how to do this, go back to Exercise 46 and do it all over again.

Installing `flask`

Before creating your first web application, you'll first need to install the "web framework" called `flask`. The term "framework" generally means "some package that makes it easier for me to do something." In the world of web applications, people create "web frameworks" to compensate for the difficult problems they've encountered when making their own sites. They share these common solutions in the form of a package you can download to bootstrap your own projects.

In our case, we'll be using the `flask` framework, but there are many, many, *many* others you can choose from. For now, learn `flask`, then branch out to another one when you're ready (or just keep using `flask` since it's good enough).

Using `pip`, install `flask`:

```
$ sudo pip install flask
[sudo] password for zedshaw:
Downloading/unpacking flask
  Running setup.py egg_info for package flask

Installing collected packages: flask
  Running setup.py install for flask

Successfully installed flask
Cleaning up...
```

This will work on Linux and macOS computers, but on Windows just drop the `sudo` part of the `pip install` command and it should work. If not, go back to Exercise 46 and make sure you can do it reliably.

Make a Simple "Hello World" Project

Now you're going to make an initial very simple "Hello World" web application and project directory using `flask`. First, make your project directory:

```
$ cd projects
$ mkdir gothonweb
$ cd gothonweb
$ mkdir bin gothonweb tests docs templates
$ touch gothonweb/__init__.py
$ touch tests/__init__.py
```

You'll be taking the game from Exercise 43 and making it into a web application, so that's why you're calling it `gothonweb`. Before you do that, we need to create the most basic `flask` application possible. Put the following code into `app.py`:

ex50.py

```
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def hello_world():
6      greeting = "World"
7      return f'Hello, {greeting}!'
8
9  if __name__ == "__main__":
10     app.run()
```

Then run the application like this:

```
(lpthw) $ python3.6 app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Finally, use your web browser and go to `http://localhost:5000/`, and you should see two things. First, in your browser you'll see `Hello, world!`. Second, you'll see your Terminal with new output like this:

```
(lpthw) $ python3.6 app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [22/Feb/2017 14:28:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [22/Feb/2017 14:28:50] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [22/Feb/2017 14:28:50] "GET /favicon.ico HTTP/1.1" 404 -
```

Those are log messages that `flask` prints out so you can see that the server is working and what the browser is doing behind the scenes. The log messages help you debug and figure out when you have problems. For example, it's saying that your browser tried to get `/favicon.ico` but that file didn't exist, so it returned the `404 Not Found` status code.

I haven't explained the way *any* of this web stuff works yet, because I want to get you set up and ready to roll so that I can explain it better in the next two exercises. To accomplish this, I'll have you break your `flask` application in various ways and then restructure it so that you know how it's set up.

What's Going On?

Here's what's happening when your browser hits your application:

1. Your browser makes a network connection to your own computer, which is called `localhost` and is a standard way of saying "whatever my own computer is called on the network." It also uses port 5000.
2. Once it connects, it makes an HTTP request to the `app.py` application and asks for the `/` URL, which is commonly the first URL on any website.
3. Inside `app.py` you've got a list of URLs and what functions they match. The only one we have is the `'/', 'index'` mapping. This means that whenever someone goes to `/` with a browser, `flask` will find the `def index` and run it to handle the request.
4. Now that `flask` has found `def index`, it calls it to actually handle the request. This function runs and simply returns a string for what `flask` should send to the browser.
5. Finally, `flask` has handled the request and sends this response to the browser, which is what you are seeing.

Make sure you really understand this. Draw up a diagram of how this information flows from your browser, to `flask`, then to `def index` and back to your browser.

Fixing Errors

First, delete line 5 where you assign the `greeting` variable, then hit refresh in your browser. Then use CTRL-C to kill `flask` and start it again. Once it's running again refresh your browser, and you should see an "Internal Server Error." Back in your Terminal you'll see this ([VENV] is the path to your `.venvs/` directory):

```
(lpthw) $ python3.6 app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
[2017-02-22 14:35:54,256] ERROR in app: Exception on / [GET]
Traceback (most recent call last):
  File "[VENV]/site-packages/flask/app.py",
    line 1982, in wsgi_app
    response = self.full_dispatch_request()
  File "[VENV]/site-packages/flask/app.py",
    line 1614, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "[VENV]/site-packages/flask/app.py",
    line 1517, in handle_user_exception
    reraise(exc_type, exc_value, tb)
  File "[VENV]/site-packages/flask/_compat.py",
  line 33, in reraise
    raise value
```

```
  File "[VENV]/site-packages/flask/app.py",
line 1612, in full_dispatch_request
    rv = self.dispatch_request()
  File "[VENV]/site-packages/flask/app.py",
line 1598, in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
  File "app.py", line 8, in index
    return render_template("index.html", greeting=greeting)
NameError: name 'greeting' is not defined
127.0.0.1 - - [22/Feb/2017 14:35:54] "GET / HTTP/1.1" 500 -
```

This works well enough, but you can also run `flask` in "debugger mode." This will give you a better error page and more useful information. The problem with debugger mode is it's not safe to run on the internet, so you have to explicitly turn it on like this:

```
(lpthw) $ export FLASK_DEBUG=1
(lpthw) $ python3.6 app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 222-752-342
```

After this, you hit refresh in your browser, and you get a much more detailed page with information you can use to debug the application and a live console to work with to find out more.

WARNING! It's the `flask` live debugging console and the improved output that makes debug mode so dangerous on the internet. With this information an attacker can completely control your machine remotely. If you ever do place your web application on the internet do *not* activate debugger mode. In fact, I would avoid making FLASK_DEBUG easy to activate. It's tempting to simply hack this startup so that you save a step during development, but then that hack will get onto your web server and it'll turn into a real hack, not just something lazy you did one night when you were tired.

Create Basic Templates

You can break your `flask` application, but did you notice that "Hello World" isn't a very good HTML page? This is a web application, and as such it needs a proper HTML response. To do that you will create a simple template that says "Hello World" in a big green font.

The first step is to create a `templates/index.html` file that looks like this:

`index.html`

```
<html>
    <head>
```

```
        <title>Gothons Of Planet Percal #25</title>
    </head>
<body>

{% if greeting %}
    I just wanted to say
    <em style="color: green; font-size: 2em;">{{ greeting }}</em>.
{% else %}
    <em>Hello</em>, world!
{% endif %}

</body>
</html>
```

If you know what HTML is, then this should look fairly familiar. If not, research HTML and try writing a few web pages by hand so you know how it works. This HTML file, however, is a template, which means that `flask` will fill in "holes" in the text depending on variables you pass in to the template. Every place you see `{{ greeting }}` will be a variable you'll pass to the template that alters its contents.

To make your `app.py` do this, you need to add some code to tell `flask` where to load the template and to render it. Take that file and change it like this:

app.py

```
1  from flask import Flask
2  from flask import render_template
3
4  app = Flask(__name__)
5
6  @app.route("/")
7  def index():
8      greeting = "Hello World"
9      return render_template("index.html", greeting=greeting)
10
11 if __name__ == "__main__":
12     app.run()
```

Once you have that in place, reload the web page in your browser, and you should see a different message in green. You should also be able to do a `View Source` on the page in your browser to see that it is valid HTML.

This may have flown by you very fast, so let me explain how a template works:

1. In your `app.py` you've imported a new function named `render_template` at the top.
2. This `render_template` knows how to load `.html` files out of the `templates/` directory, because that is the default magic setting for a `flask` application.

3. Later in your code, when the browser hits the `def index`, instead of just returning the string `greeting`, you call `render_template` and pass the greeting to it as a variable.
4. This `render_template` method then loads the `templates/index.html` file (even though you didn't explicitly say `templates`) and processes it.
5. In this `templates/index.html` file you have what looks like normal HTML, but then there's "code" placed between two kinds of markers. One is `{% %}`, which marks pieces of "executable code" (if-statements, for-loops, etc.). The other is `{{ }}`, which marks variables to be converted into text and placed into the HTML output. The `{% %}` executable code doesn't show up in the HTML. To learn more about this template language read the Jinja2 documentation.

To get deeper into this, change the greeting variable and the HTML to see what effect it has. Also create another template named `templates/foo.html` and render that like before.

I think Ansible uses this too

Study Drills

1. Read the documentation at http://flask.pocoo.org/docs/0.12/, which is the same as the `flask` project.
2. Experiment with everything you can find there, including their example code.
3. Read about HTML5 and CSS3 and make some other .html and .css files for practice.
4. If you have a friend who knows Django and is willing to help you, then consider doing Exercises 50, 51, and 52 in Django instead to see what that's like.

I might take him up on this.

Common Student Questions

I can't seem to connect to `http://localhost:5000/`. Try `http://127.0.0.1:5000/` instead.

I can't find `index.html` **(or just about anything).** You probably are doing `cd bin/` first and then trying to work with the project. Do not do this. All of the commands and instructions assume you are one directory above `bin/`, so if you can't type `python3.6 app.py` then you are in the wrong directory.

Why do we assign `greeting=greeting` **when we call the template?** You are not assigning to `greeting`. You are setting a named parameter to give to the template. It's sort of an assignment, but it only affects the call to the template function.

I can't use port 5000 on my computer. You probably have an anti-virus program installed that is using that port. Try a different port.

EXERCISE 51

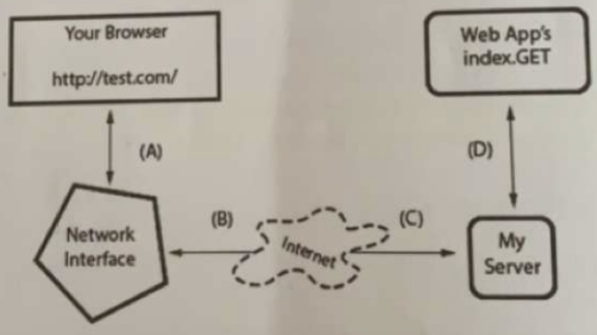
Getting Input from a Browser

While it's exciting to see the browser display "Hello World," it's even more exciting to let the user submit text to your application from a form. In this exercise we'll improve our starter web application by using forms and storing information about users into their "sessions."

How the Web Works

Time for some boring stuff. You need to understand a bit more about how the web works before you can make a form. This description isn't complete, but it's accurate and will help you figure out what might be going wrong with your application. Also, creating forms will be easier if you know what they do.

I'll start with a simple diagram that shows you the different parts of a web request and how the information flows:



I've labeled the lines with letters so I can walk you through a regular request process:

1. You type in the url http://test.com// into your browser, and it sends the request on `line (A)` to your computer's network interface.
2. Your request goes out over the internet on `line (B)` and then to the remote computer on `line (C)` where my server accepts the request.
3. Once my computer accepts it, my web application gets it on `line (D)`, and my Python code runs the `index.GET` handler.
4. The response comes out of my Python server when I `return` it, and it goes back to your browser over `line (D)` again.

5. The server running this site takes the response off `line (D)`, then sends it back over the internet on `line (C)`.
6. The response from the server then comes off the internet on `line (B)`, and your computer's network interface hands it to your browser on `line (A)`.
7. Finally, your browser then displays the response.

In this description there are a few terms you should know so that you have a common vocabulary to work with when talking about your web application:

Browser The software that you're probably using every day. Most people don't know what a browser really does. They just call browsers "the internet." Its job is to take addresses (like http://test.com/) you type into the URL bar, then use that information to make requests to the server at that address.

Address This is normally a URL (Uniform Resource Locator) like http://test.com/ and indicates where a browser should go. The first part, `http`, indicates the protocol you want to use, in this case "Hyper-Text Transport Protocol." You can also try ftp://ibiblio.org/ to see how "File Transport Protocol" works. The http://test.com/ part is the "hostname," a human readable address you can remember and which maps to a number called an IP address, similar to a telephone number for a computer on the internet. Finally, URLs can have a trailing `path` like the `/book/` part of http://test.com/book/, which indicates a file or some resource *on* the server to retrieve with a request. There are many other parts, but those are the main ones.

Connection Once a browser knows what protocol you want to use (http), what server you want to talk to (http://test.com/), and what resource on that server to get, it must make a connection. The browser simply asks your operating system (OS) to open a "port" to the computer, usually port 80. When it works, the OS hands back to your program something that works like a file, but is actually sending and receiving bytes over the network wires between your computer and the other computer at http://test.com/. This is also the same thing that happens with http://localhost:8080/, but in this case you're telling the browser to connect to your own computer (localhost) and use port 8080 rather than the default of 80. You could also do http://test.com:80/ and get the same result, except you're explicitly saying to use port 80 instead of letting it be that by default.

Request Your browser is connected using the address you gave. Now it needs to ask for the resource it wants (or you want) on the remote server. If you gave `/book/` at the end of the URL, then you want the file (resource) at `/book/`, and most servers will use the real file `/book/index.html` but pretend it doesn't exist. What the browser does to get this resource is send a *request* to the server. I won't get into exactly how it does this, but just understand that it has to send something to query the server for the request. The interesting thing is that these "resources" don't have to be files. For instance, when the browser in your application asks for something, the server is returning something your Python code generated.

Server The server is the computer at the end of a browser's connection that knows how to answer your browser's requests for files/resources. Most web servers just send files, and that's actually the majority of traffic. But you're actually building a server in Python that knows how to take requests for resources and then return strings that you craft using Python. When you do this crafting, you are pretending to be a file to the browser, but really it's just code. As you can see from Exercise 50, it also doesn't take much code to create a response.

Response This is the HTML (CSS, JavaScript, or images) your server wants to send back to the browser as the answer to the browser's request. In the case of files, it just reads them off the disk and sends them to the browser, but it wraps the contents of the disk in a special "header" so the browser knows what it's getting. In the case of your application, you're still sending the same thing, including the header, but you generate that data on the fly with your Python code.

That is the fastest crash course in how a web browser accesses information on servers on the internet. It should work well enough for you to understand this exercise, but if not, read about it as much as you can until you get it. A really good way to do that is to take the diagram and break different parts of the web application you did in Exercise 50. If you can break your web application in predictable ways using the diagram, you'll start to understand how it works.

How Forms Work

The best way to play with forms is to write some code that accepts form data, and then see what you can do. Take your `app.py` file and make it look like this:

`form_test.py`

```
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)

@app.route("/hello")
def index():
    name = request.args.get('name', 'Nobody')

    if name:
        greeting = f"Hello, {name}"
    else:
        greeting = "Hello World"

    return render_template("index.html", greeting=greeting)

if __name__ == "__main__":
    app.run()
```

Restart it (hit CTRL-C and then run it again) to make sure it loads again, then with your browser go to `http://localhost:5000/hello`, which should display, "I just wanted to say Hello, Nobody." Next, change the URL in your browser to `http://localhost:5000/hello?name=Frank`, and you'll see it say, "Hello, Frank." Finally, change the `name=Frank` part to be your name. Now it's saying hello to you.

Let's break down the changes I made to your script.

1. Instead of just a string for `greeting`, I'm now using `request.args` to get data from the browser. This is a simple `dict` that contains the form values as key=value pairs.
2. I then construct the `greeting` from the new `name`, which should be very familiar to you by now.
3. Everything else about the file is the same as before.

You're also not restricted to just one parameter on the URL. Change this example to give two variables like this: `http://localhost:5000/hello?name=Frank&greet=Hola`. Then change the code to get `name` and `greet` like this:

```
greet = request.args.get('greet', 'Hello')
greeting = f"{greet}, {name}"
```

You should also try *not* giving the `greet` and `name` parameters on the URL. You'll simply send your browser to `http://localhost:5000/hello` to see that the `index` now defaults to "Nobody" for `name` and "Hello" for `greet`.

Creating HTML Forms

Passing the parameters on the URL works, but it's kind of ugly and not easy to use for regular people. What you really want is a "POST form," which is a special HTML file that has a `<form>` tag in it. This form will collect information from the user, then send it to your web application just like you did above.

Let's make a quick one so you can see how it works. Here's the new HTML file you need to create, in `templates/hello_form.html`:

`hello_form.html`

```
<html>
    <head>
        <title>Sample Web Form</title>
    </head>
<body>

<h1>Fill Out This Form</h1>
```

```
    <form action="/hello" method="POST">
        A Greeting: <input type="text" name="greet">
        <br/>
        Your Name: <input type="text" name="name">
        <br/>
        <input type="submit">
    </form>

    </body>
    </html>
```

You should then change `app.py` to look like this:

app.py

```
1   from flask import Flask
2   from flask import render_template
3   from flask import request
4
5   app = Flask(__name__)
6
7   @app.route("/hello", methods=['POST', 'GET'])
8   def index():
9       greeting = "Hello World"
10
11      if request.method == "POST":
12          name = request.form['name']
13          greet = request.form['greet']
14          greeting = f"{greet}, {name}"
15          return render_template("index.html", greeting=greeting)
16      else:
17          return render_template("hello_form.html")
18
19
20  if __name__ == "__main__":
21      app.run()
```

Once you've got those written up, simply restart the web application again and hit it with your browser like before.

This time you'll get a form asking you for "A Greeting" and "Your Name." When you hit the `Submit` button on the form, it will give you the same greeting you normally get, but this time look at the URL in your browser. See how it's `http://localhost:5000/hello` even though you sent in parameters.

The part of the `hello_form.html` file that makes this work is the line with `<form action="/hello" method="POST">`. This tells your browser to:

1. Collect data from the user using the form fields inside the form.
2. Send them to the server using a POST type of request, which is just another browser request that "hides" the form fields. Safer than Get
3. Send that to the `/hello` URL (as shown in the `action="/hello"` part).

You can then see how the two <input> tags match the names of the variables in your new code. Also notice that instead of just a GET method inside class index, I have another method, POST. How this new application works is as follows:

1. Your request goes to index() like normal, except now there is an if-statement that checks the request.method for either "POST" or "GET" methods. This is how the browser tells app.py that a request is either a form submission or URL parameters.
2. If request.method is "POST", then you process the form as if it were filled out and submitted, returning the proper greeting.
3. If request.method is anything else, then you simply return the hello_form.html for the user to fill out.

As an exercise, go into the templates/index.html file and add a link back to just /hello so that you can keep filling out the form and seeing the results. Make sure you can explain how this link works and how it's letting you cycle between templates/index.html and templates/hello_form.html and what's being run inside this latest Python code.

Creating a Layout Template

When you work on your game in the next exercise, you'll need to make a bunch of little HTML pages. Writing a full web page each time will quickly become tedious. Luckily you can create a "layout" template, or a kind of shell that will wrap all your other pages with common headers and footers. Good programmers try to reduce repetition, so layouts are essential for being a good programmer.

Change templates/index.html to be like this:

index_laid_out.html

```
{% extends "layout.html" %}

{% block content %}

{% if greeting %}
    I just wanted to say
    <em style="color: green; font-size: 2em;">{{ greeting }}</em>.
{% else %}
    <em>Hello</em>, world!
{% endif %}

{% endblock %}
```

Then change templates/hello_form.html to be like this:

hello_form_laid_out.html

```
{% extends "layout.html" %}

{% block content %}
```

```
    <h1>Fill Out This Form</h1>

    <form action="/hello" method="POST">
        A Greeting: <input type="text" name="greet">
        <br/>
        Your Name: <input type="text" name="name">
        <br/>
        <input type="submit">
    </form>

    {% endblock %}
```

All we're doing is stripping out the "boilerplate" at the top and the bottom, which is always on every page. We'll put that back into a single `templates/layout.html` file that handles it for us from now on.

Once you have those changes, create a `templates/layout.html` file with this in it:

`layout.html`

```
<html>
<head>
    <title>Gothons From Planet Percal #25</title>
</head>
<body>

{% block content %}

{% endblock %}

</body>
</html>
```

This file looks like a regular template, except that it's going to be passed the *contents* of the other templates and used to *wrap* them. Anything you put in here doesn't need to be in the other templates. Your other HTML templates will be inserted into the `{% block content %}` section. `flask` knows to use this `layout.html` as the layout because you put `{% extends "layout.html" %}` at the top of your templates.

Writing Automated Tests for Forms

It's easy to test a web application with your browser by just hitting refresh, but come on, we're programmers here. Why do some repetitive task when we can write some code to test our application? What you're going to do next is write a little test for your web application form based on what you learned in Exercise 47. If you don't remember Exercise 47, read it again.