Introduction To Flask

A quick way to turn your code into an app

# Introduction

Alright. You have made your code, debugged it, validated it and are very pleased with the results. In fact you are so proud of it, you think you deserve a medal. And perhaps you do!

But who will give you this medal? Who (other than you and your teacher and maybe a couple of your friends) even knows what you have done? Can you even put your work on your resume?

Yes you can. *Iff[[1]](#footnote-1)* you can put it in a format that is (a) easily accessible and (b) user friendly. In other words, you have to make your own web-app. And the Flask library of Python allows you to do just that. Aren’t you lucky! ☺

This document is based on the excellent tutorial “The Flask Megatutorial”[[2]](#footnote-2) by Miguel Grinberg[[3]](#footnote-3).

# Installation

To begin, you have to install Flask and a number of associated libraries. See this list?

Babel==2.2.0

Flask==0.10.1

Flask-Babel==0.9

Flask-Login==0.3.2

Flask-Mail==0.9.1

Flask-OpenID==1.2.5

Flask-SQLAlchemy==2.1

Flask-WTF==0.12

Flask-WhooshAlchemy==0.56

Jinja2==2.8

MarkupSafe==0.23

SQLAlchemy==1.0.12

Tempita==0.5.2

WTForms==2.1

Werkzeug==0.11.4

Whoosh==2.7.2

blinker==1.4

coverage==4.0.3

decorator==4.0.9

flipflop==1.0

guess-language==0.2

itsdangerous==0.24

numpy==1.10.4

pandas==0.17.1

pbr==1.8.1

python-dateutil==2.5.0

python-openid==2.2.5

pytz==2015.7

six==1.10.0

speaklater==1.3

sqlalchemy-migrate==0.10.0

sqlparse==0.1.18

Copy and paste the above list: Babel==2.2.0 to sqlparse==0.1.18 into a file called “requirements.txt”.

Now make a folder called: “introtoflask” on your machine. Navigate into it using Windows Explorer and open an “Enhanced Console” in it. See Figure 1.



Figure 1: Open an enhanced console (right click anywhere in the folder window) in a new folder called “introtoflask”. Note that “requirements.txt” is already present in the folder.

In the console that opens up, type the following (see Figure 2):

virtualenv flask

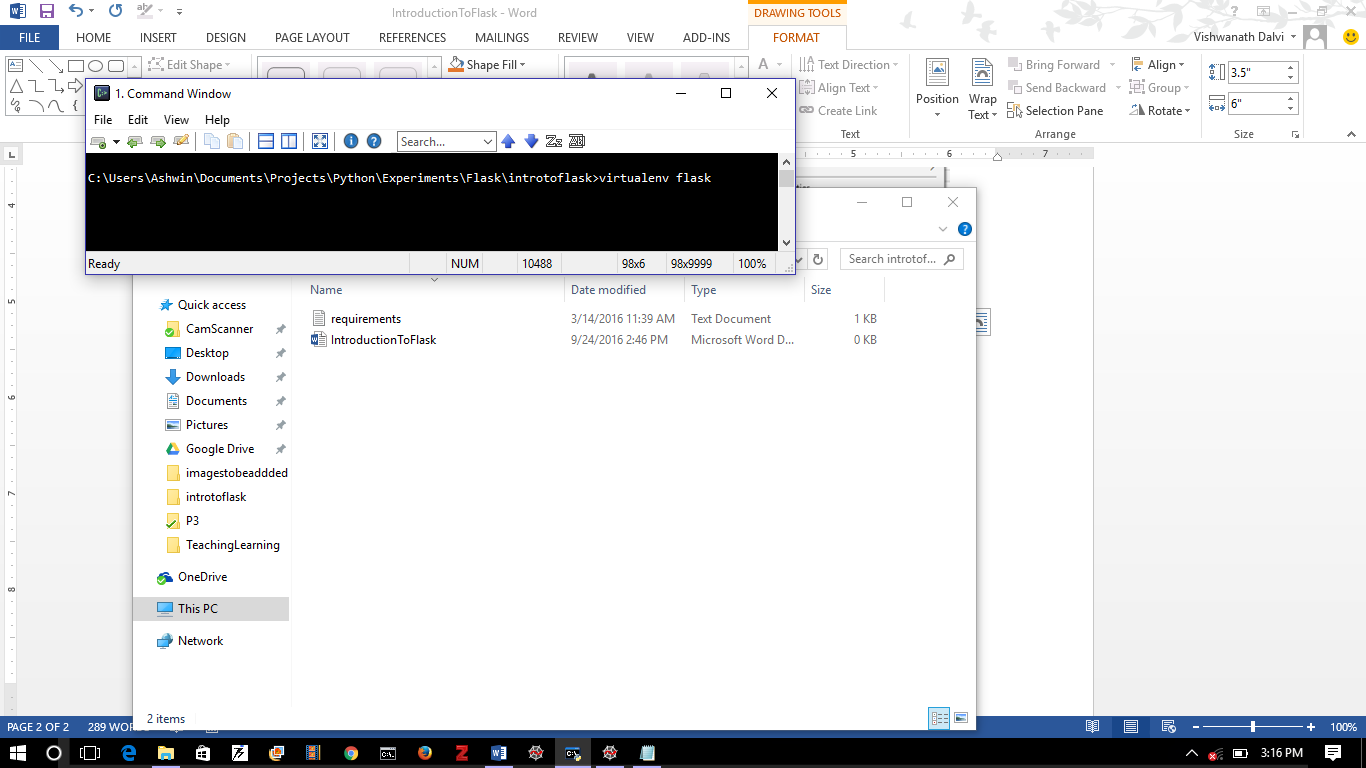


Figure 2: Making a virtual environment

Note: If you want to install all packages currently installed in your Python setup, use:

virtualenv flask --system-site-packages

Once that is done, you will see a new sub-folder called “flask” appear in the Explorer window. You now have a virtual environment set up where you can install whatever you wish without the admins breathing down your neck ☺. Now just activate it! See Figure 3.

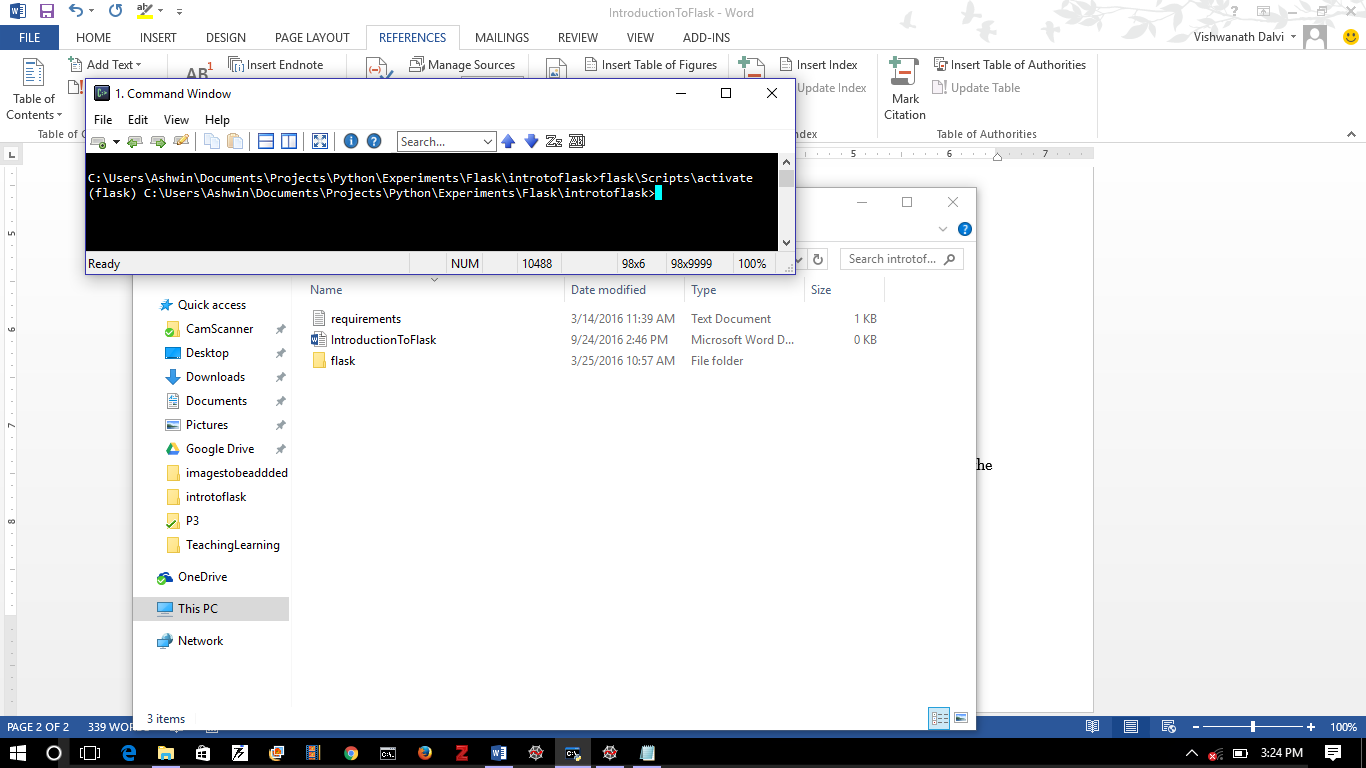


Figure 3: Activating a virtual environment. We are running an executable called “activate” located in the Scripts subdirectory of the flask subfolder. Notice how the path in the prompt is now preceded by “(flask)”.

Now we can “pip” install all the libraries listed in “requirements.txt” as shown in Figure 4

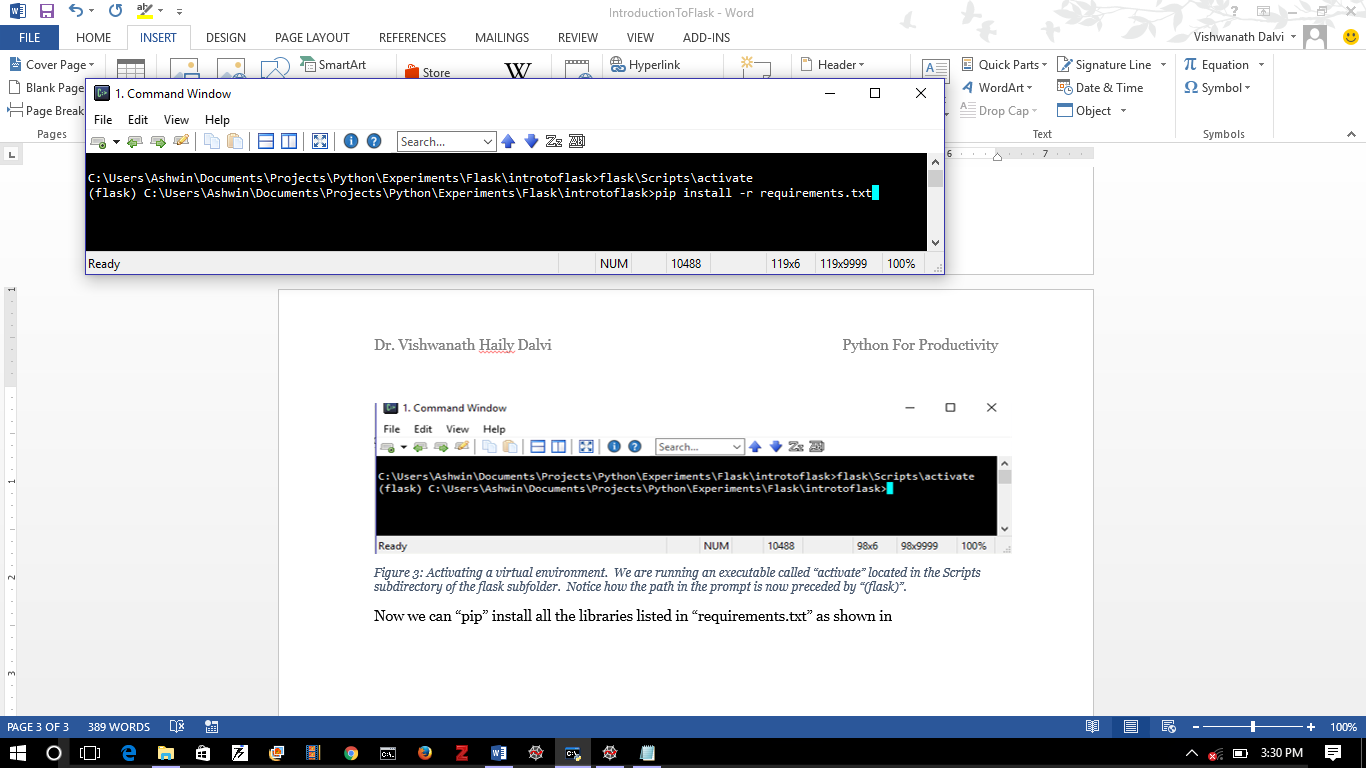


Figure 4: Installing the libraries listed in “requirements.txt” using pip.

Now you are ready to use Flask!

# Simple Web App

You first need to make a sub-folder called “app”. See Figure 5.

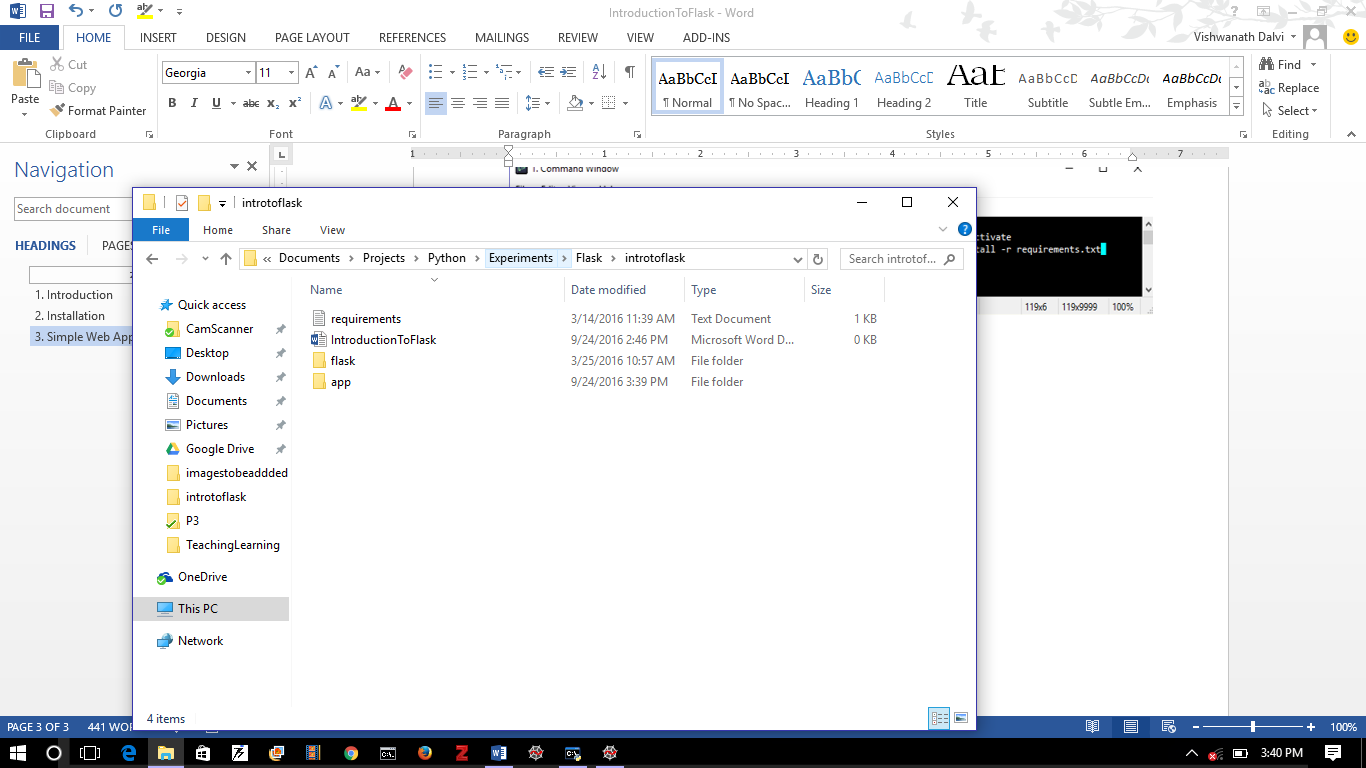


Figure 5: Make a new sub-folder called “app”.

Inside this “app” folder place a file called “\_\_init\_\_.py”. The contents of “\_\_init\_\_.py” are very simple: see Figure 6.

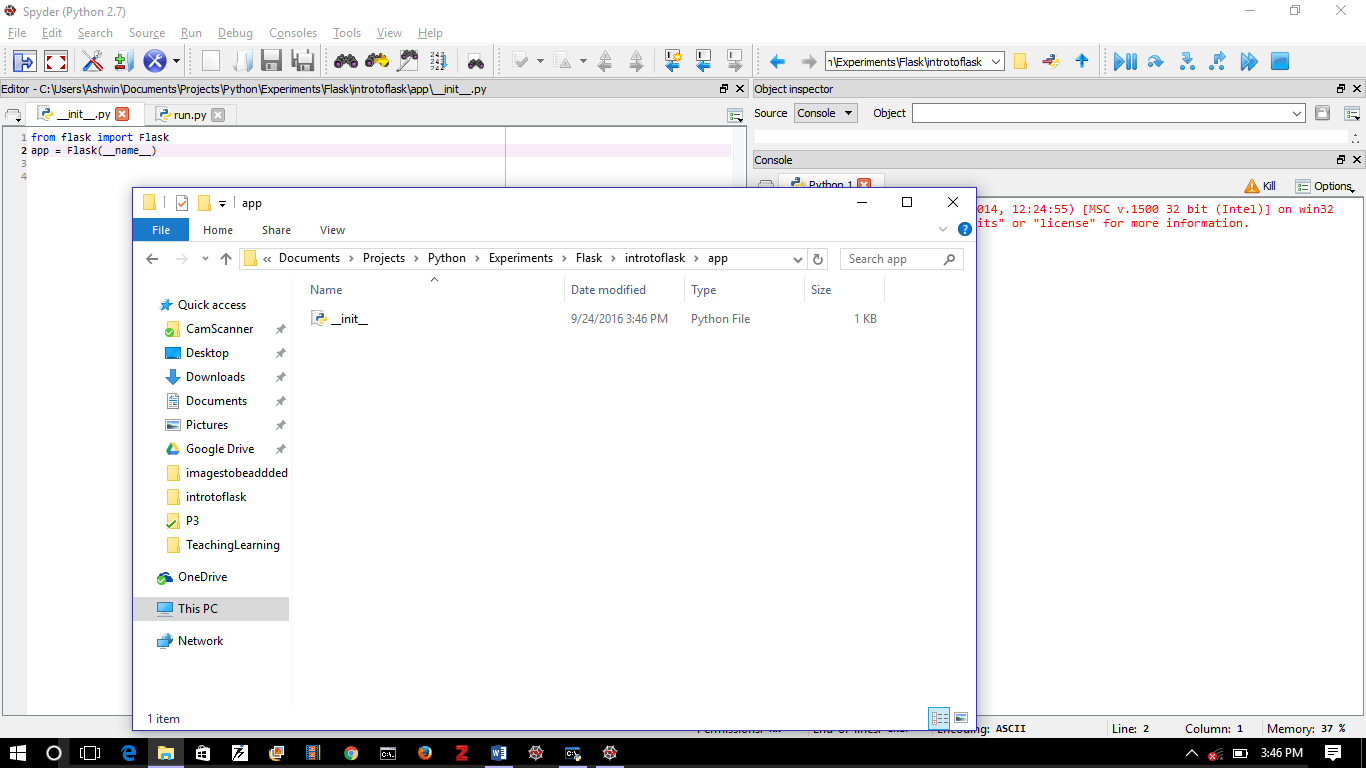
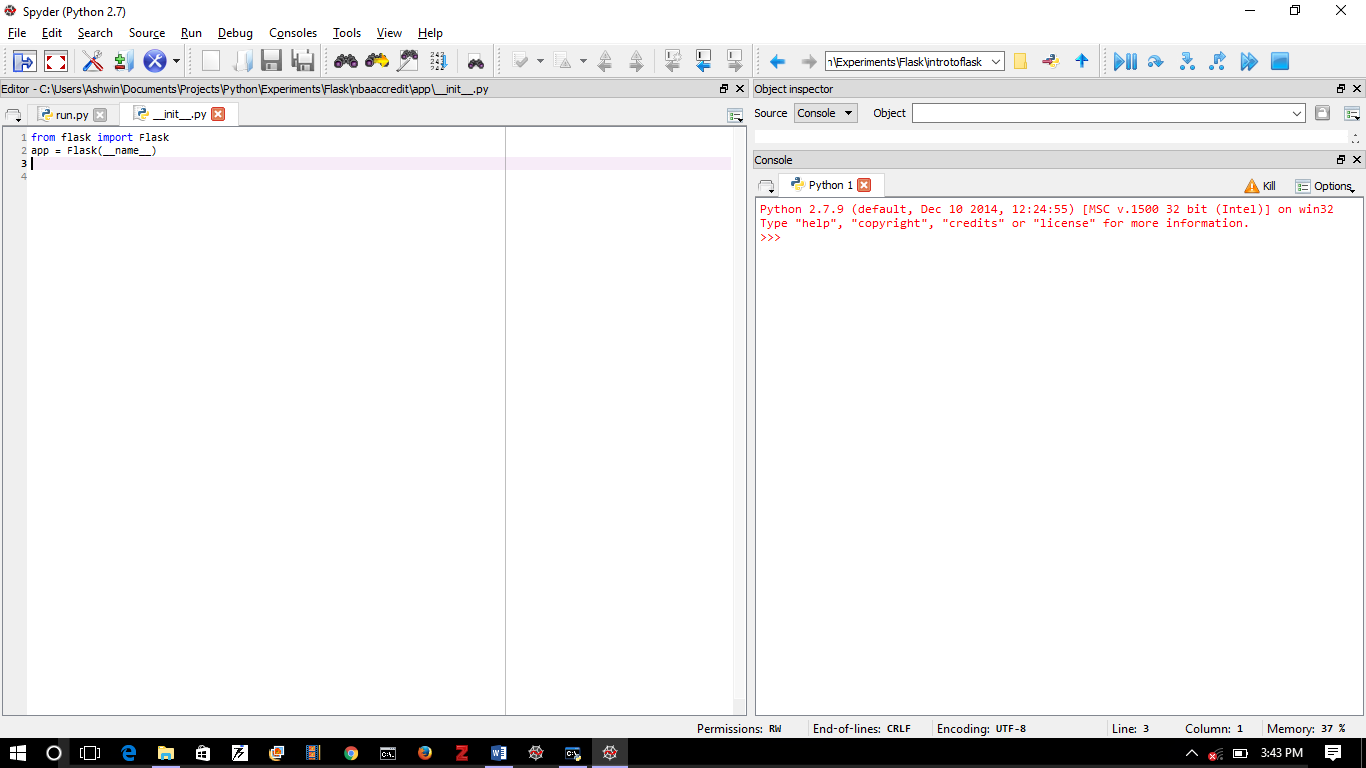


Figure 6: Adding a “ \_\_init\_\_.py” file to the “app” subfolder to turn it into a python module. Within the “\_\_init\_\_.py” file we have instantiated a Flask object and assigned it to the variable “app”.

And now we need to run it. Put the script “run.py” in the “introtoflask” folder. See Figure 7.

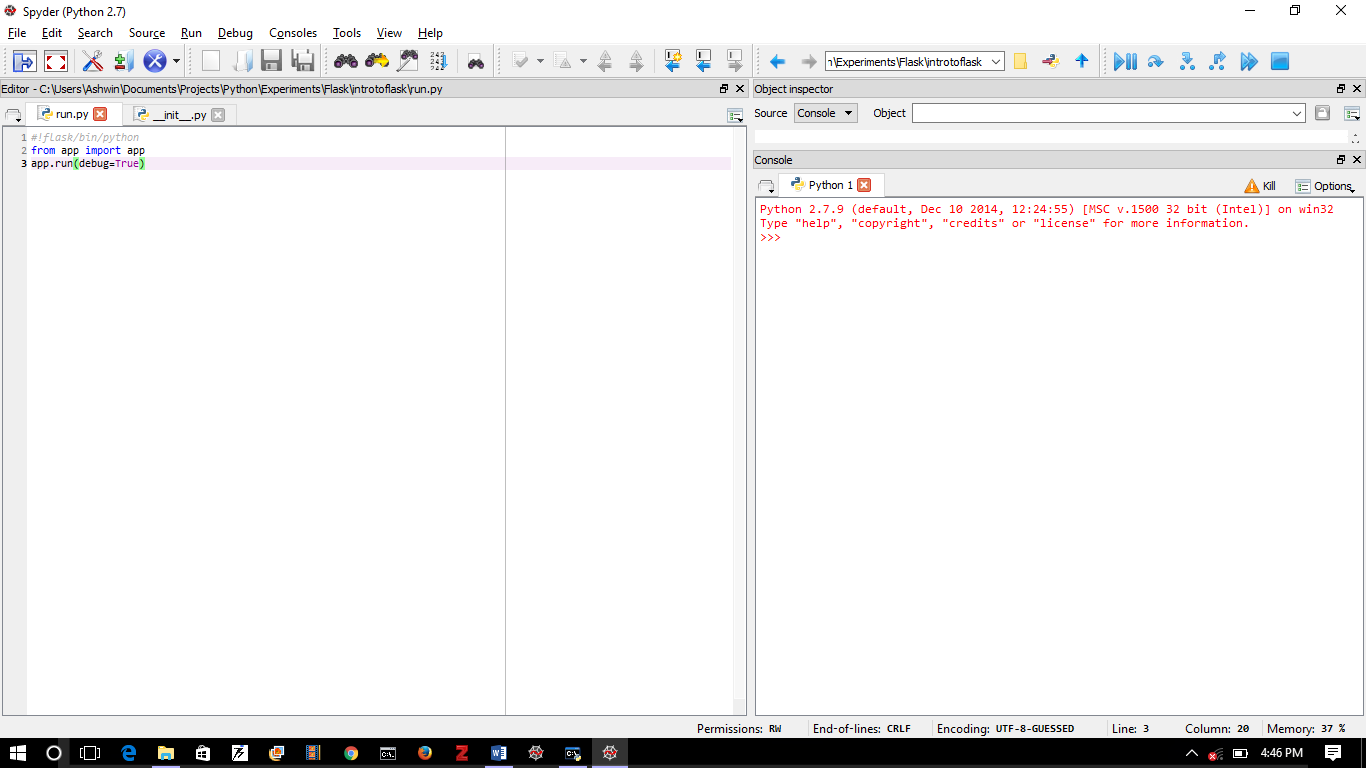
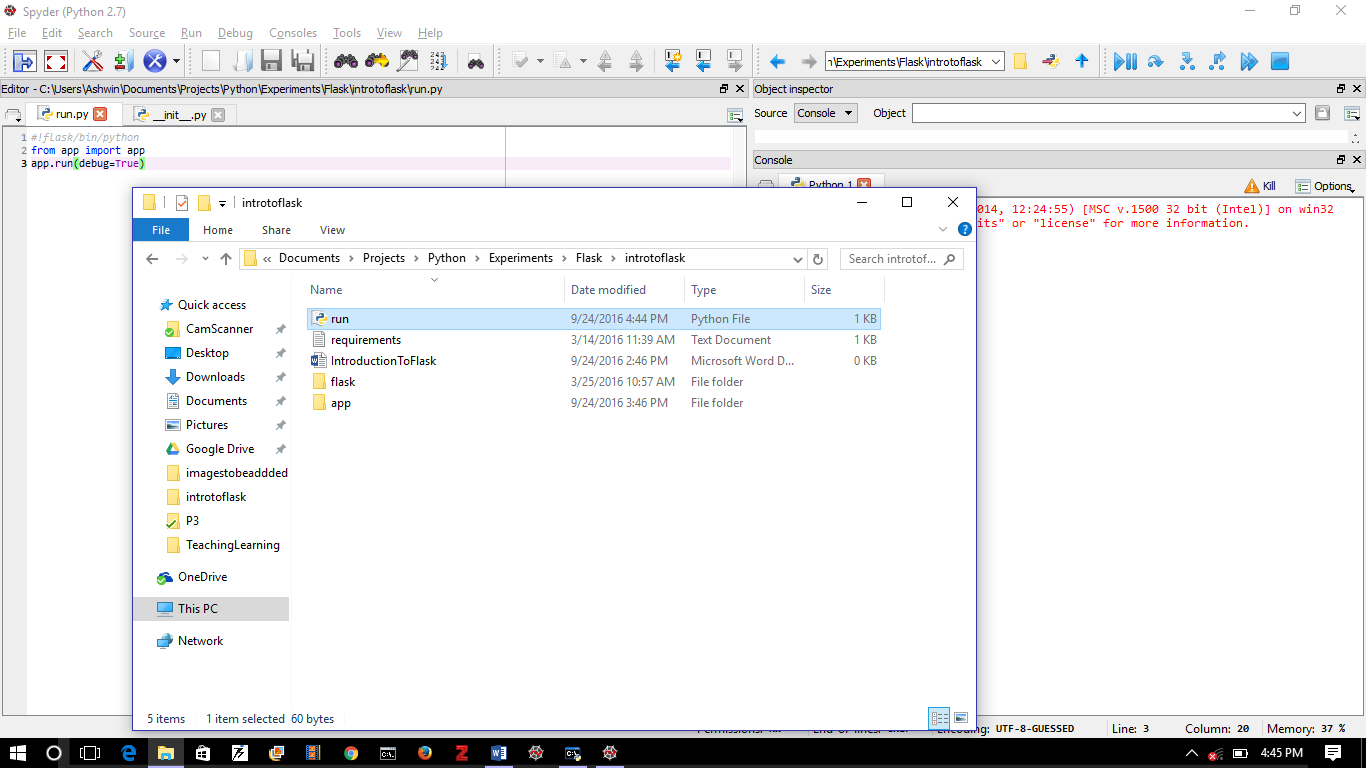


Figure 7: The “run.py” script to run the app. Note that it is in the “introtoflask” folder NOT in the “app” subfolder.

Now let’s run it! See Figure 8. If we type “flask\Scripts\python.exe run.py” in the command window and hit Enter, we see that a server is started and it listens on the url: <http://127.0.0.1:5000> which is also known as <http://localhost:5000>. But if we type this url into our browser, we find a 404 error: because we haven’t specified what to display!

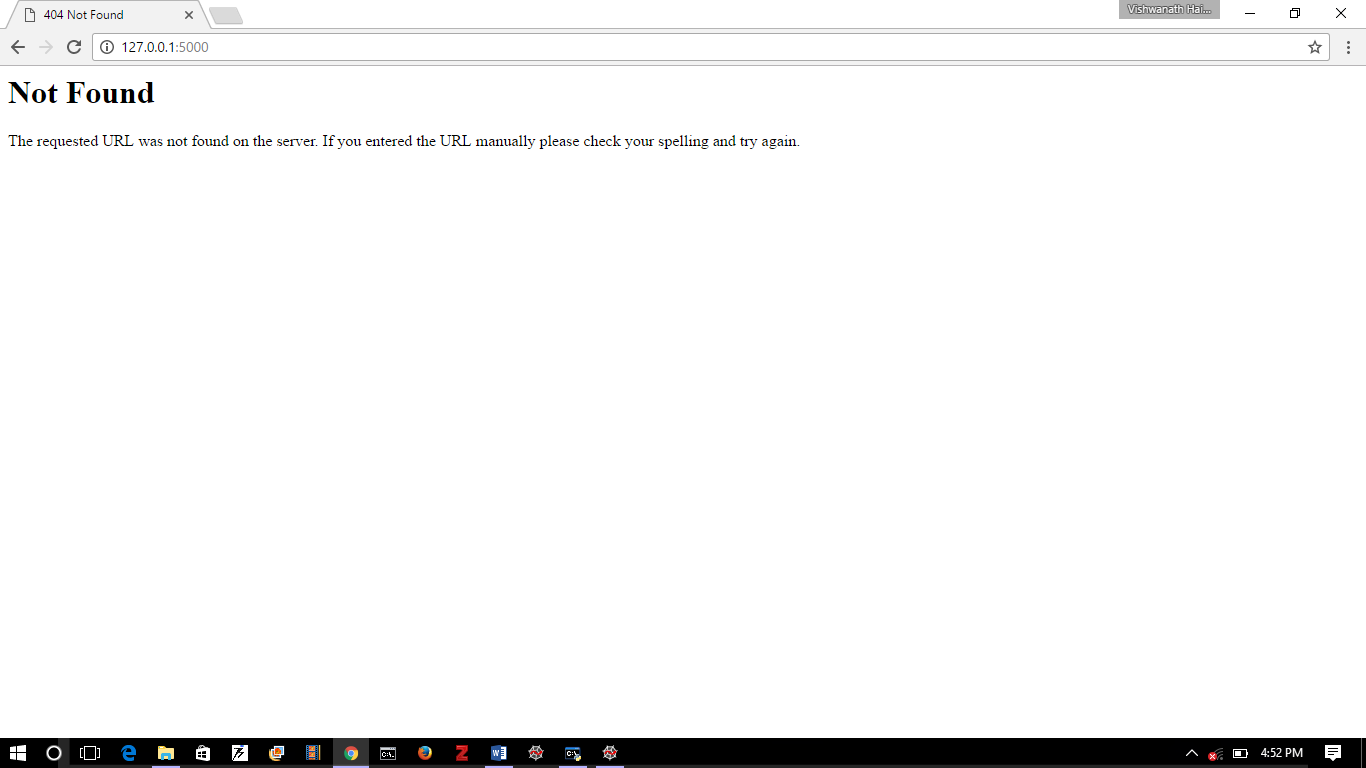
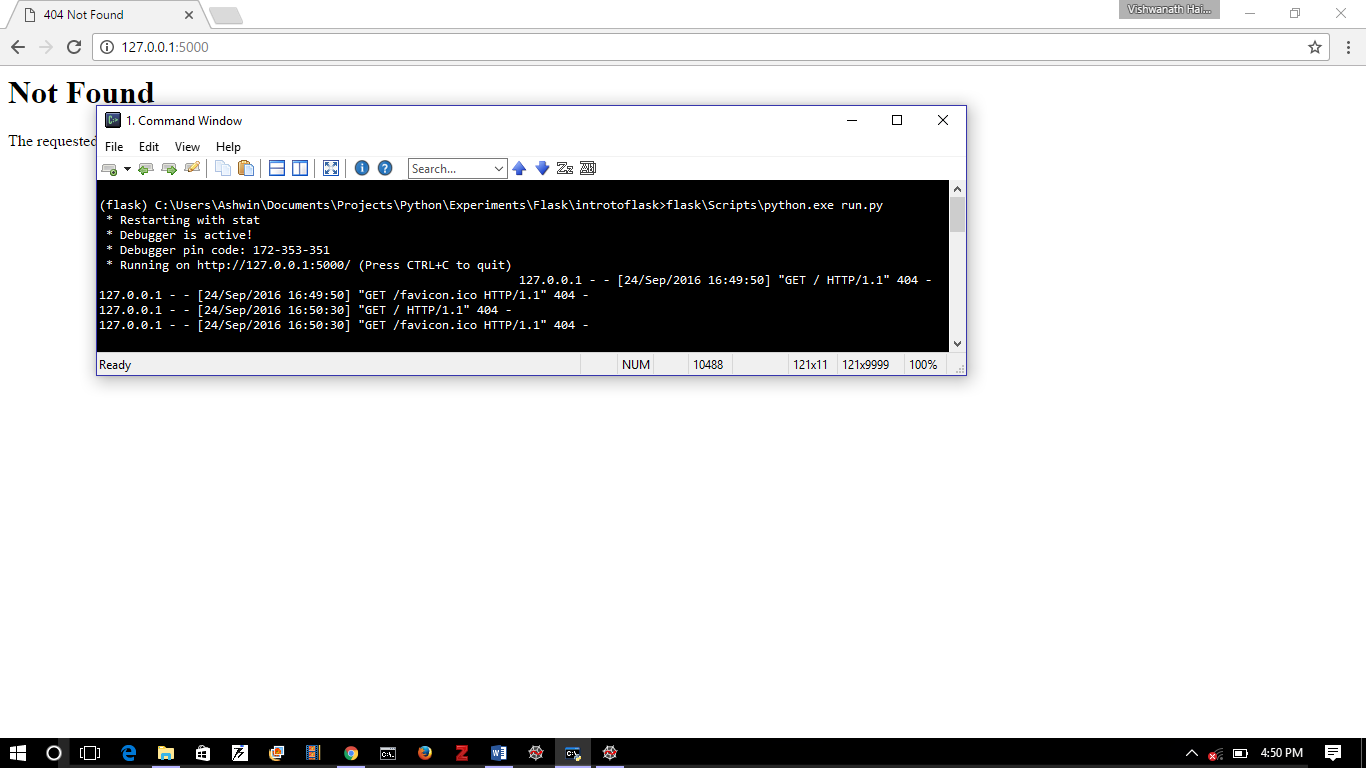


Figure 8: The script “run.py” is run by typing “flask\Scripts\python.exe run.py” in the Command Prompt. The messages tell us the server is running at port 5000 of localhost i.e. <http://127.0.0.1:5000/> If you find this hard to remembers, just use <http://localhost:5000/> . In any case, when you type the url into a browser, you will just get a “404 Not Found” error. Because we have not yet told the thing what to display!

So let’s do that!

To display anything, we need to specify an .html file (*template)* and to tell the system when and how to *render* it. So two things.

Let’s do this one at a time. Let’s begin with the *template*.

Templates are .html documents stored in a folder called “templates” that lives in the “app” folder. Let’s go ahead and create it. See Figure 9.

We will add a document called “base.html” in it. To do so, open a new file in Spyder and save it in the “templates” folder as a “.html” and NOT a “.py” file.

The “base.html” contains details for what can be thought of as the *parent class* of templates. Alternatively, to use MS-Word terminology, it contains the header/footer for the web-app: it is common for all templates.

The entire template is between <html>and</html> tags indicating that everything in between should be interpreted as a html document. The content between <head> and </head> is displayed in the browser tab. The matter between <body> and </body> is displayed in the browser window. The <div> </div> mark a division. And the <a href= “/home”> Home </a> is a *hyperlink* to the home page. The child templates that extends base.html place their content between the {% block content %} and {% endblock %}.

In the home.html template, the first line indicates the “parent” template. Hence {% extends “base.html” %}. The entire code is sandwiched between {% block content %} and {% endblock %} which matches the corresponding entry in the base.html template. The content of home.html is right now a simple text message. It has the heading between the <h1> </h1> tags (heading 1) and the main text between the paragraph tags <p> </p>. Note that .html is white-space *insensitive*.

If you want to know more about html, css, javascript etc, please visit <http://www.w3schools.com/> . It’s a wonderful resource.

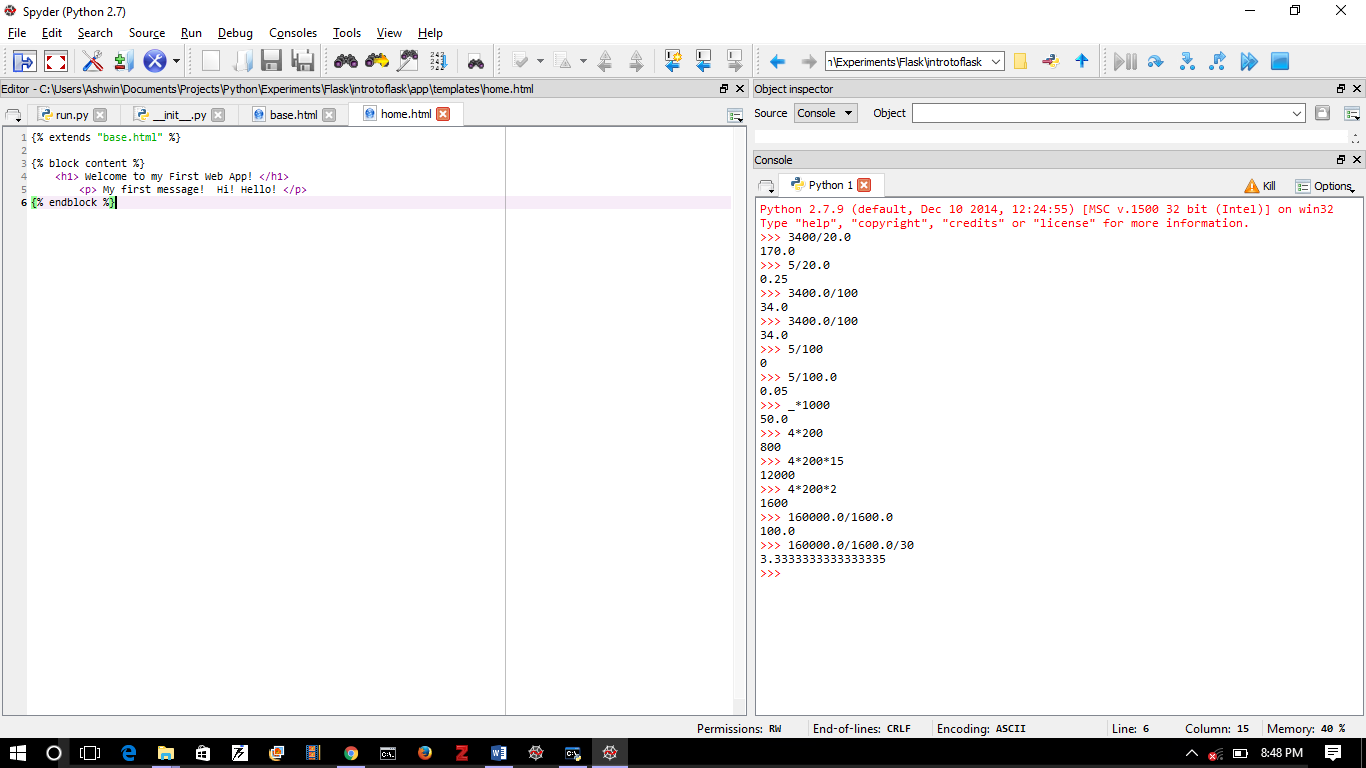
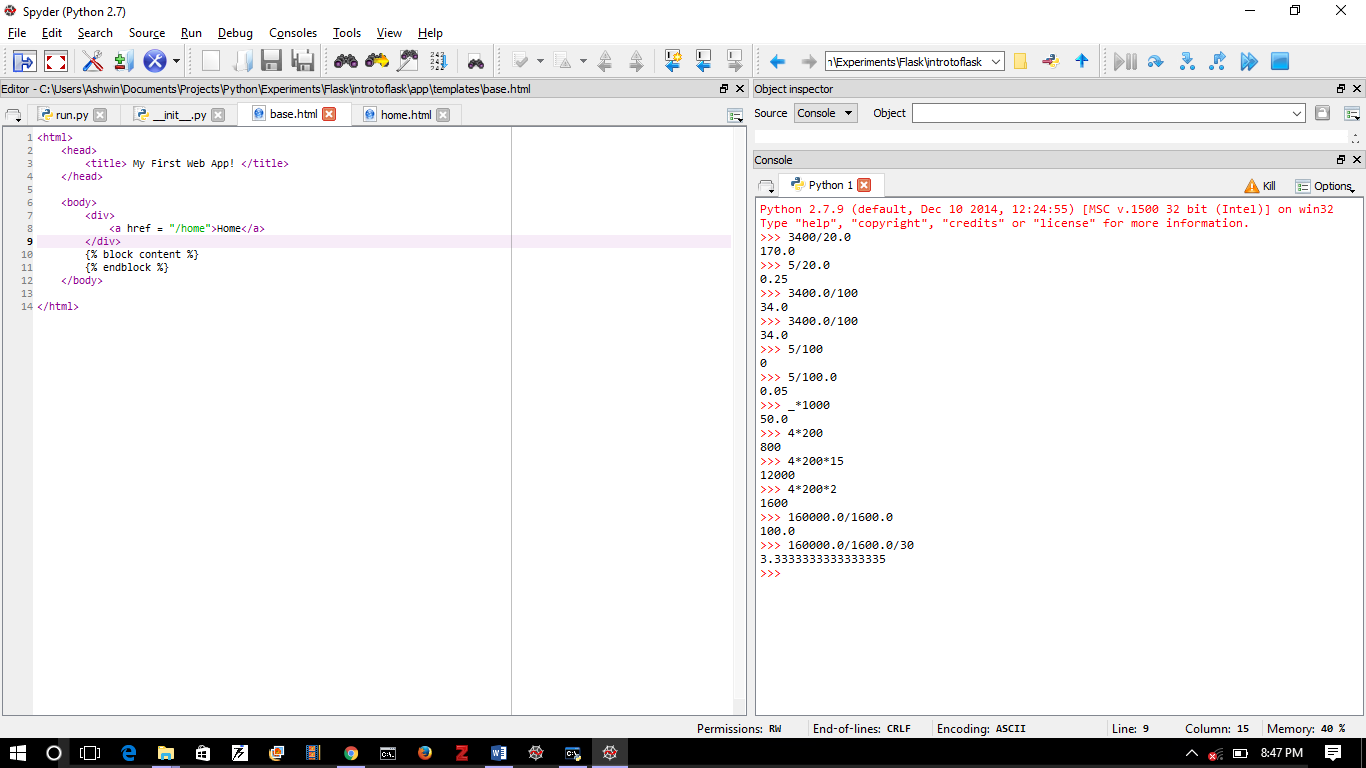
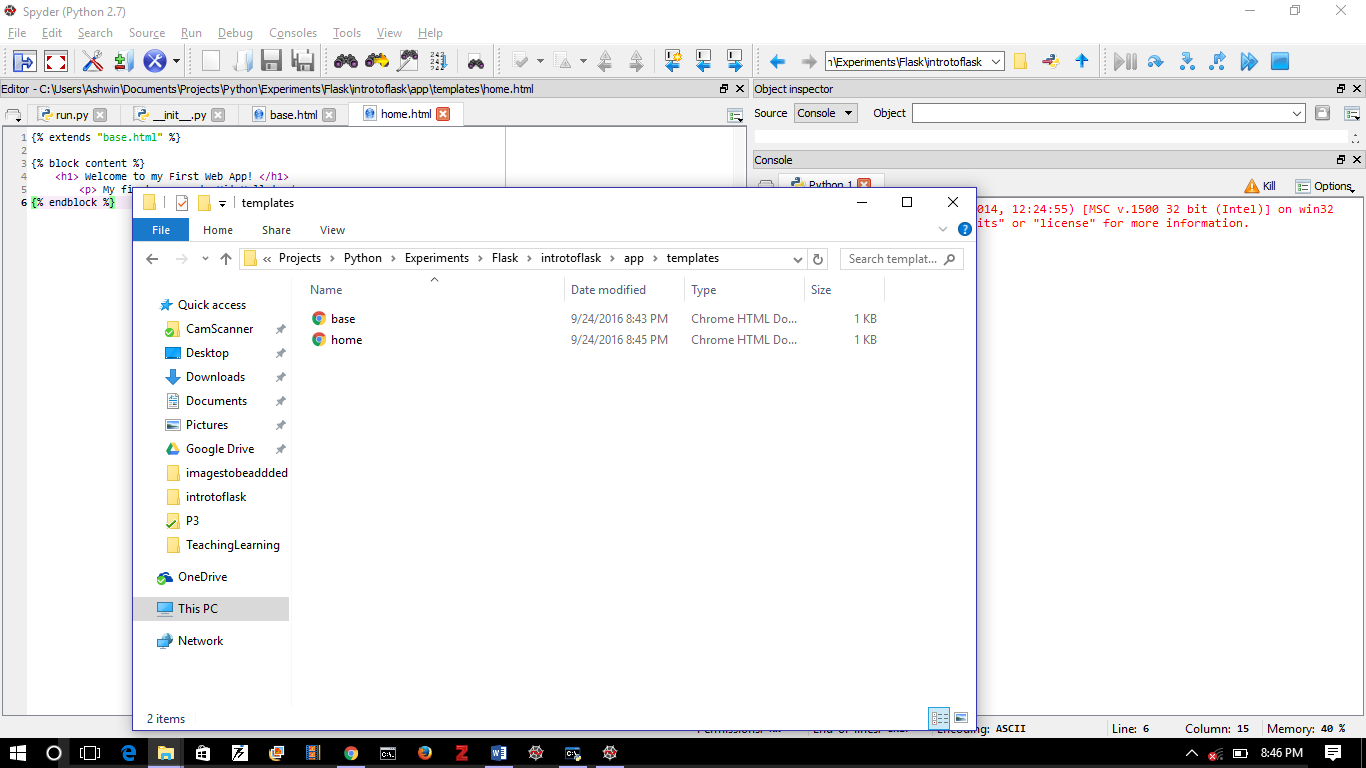
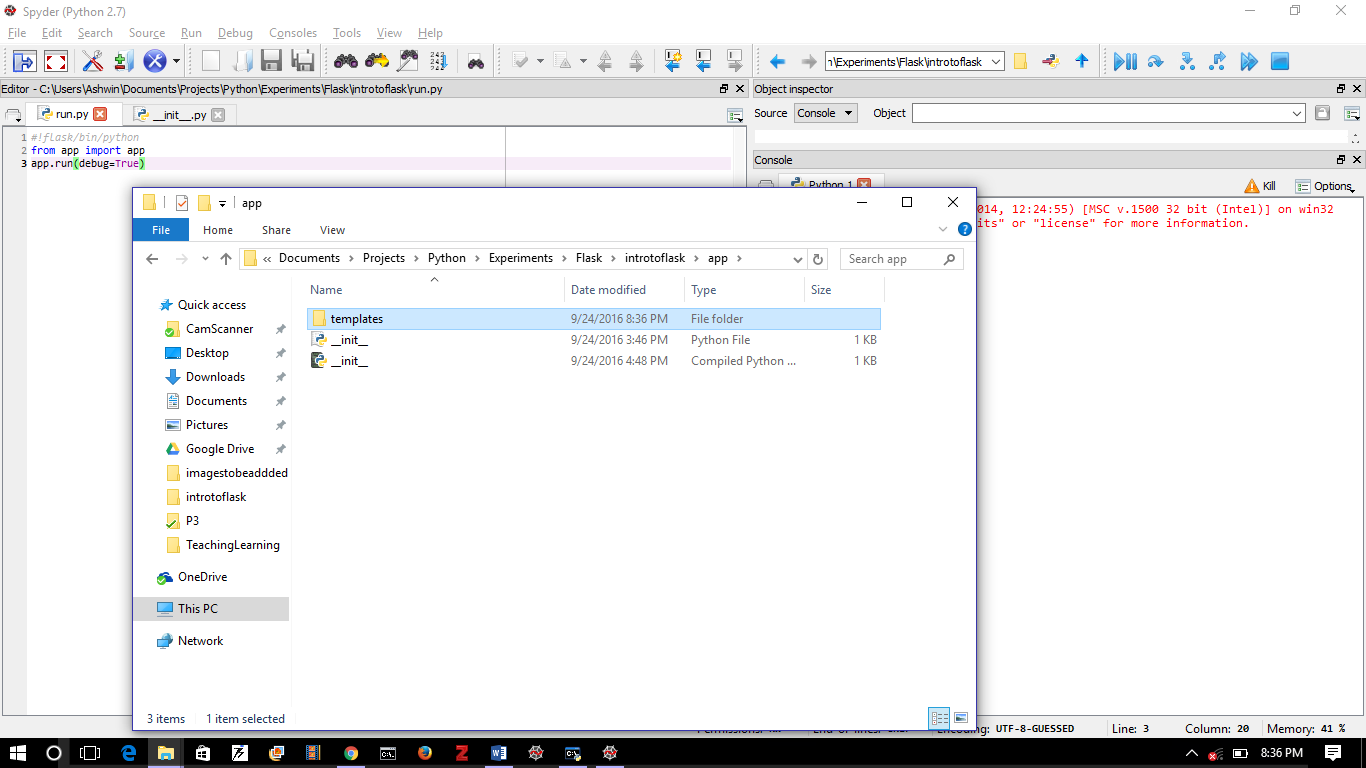


Figure 9: We have made a folder called “templates” in the folder called “app”. It has two .html templates (why they are called templates will be clear shortly). They are “base.html” and “home.html”. The contents are displayed. The “base.html” is something like a header/footer in a Word document or like a parent class: it is common to all pages. Other templates **extend** the base template. The additional content in sandwiched between {% block content %} and {% endblock %}. Notice that .html documents can be edited within Spyder itself!

But is this enough? No. We still have to tell Flask how to use the templates. That we do in another script called “views.py” that lives in the “app” folder.

The “views.py” script lives in the app folder. It is as shown in Figure 10. Don’t get too distracted by the @app.route(“/”) lines. They are called *decorators*. They *decorate* a function: in this case home. The upshot is that whenever the url for “home” (or blank) is requested by the browser, the function home is called. In our case, home does little but *render* or display the home.html template.

You can now go to <http://localhost:5000/> (or refresh it if you are already there) to see the home.html template rendered. See how the <title> </title> tagged material and the hyperlinked “Home” from base.html appears in the tab and how the <h1></h1> tagged title and <p></p> tagged text from home.html are displayed. And did you notice you did not have to restart the server? The system just took the changes ☺.

Give yourself a small pat on the back.

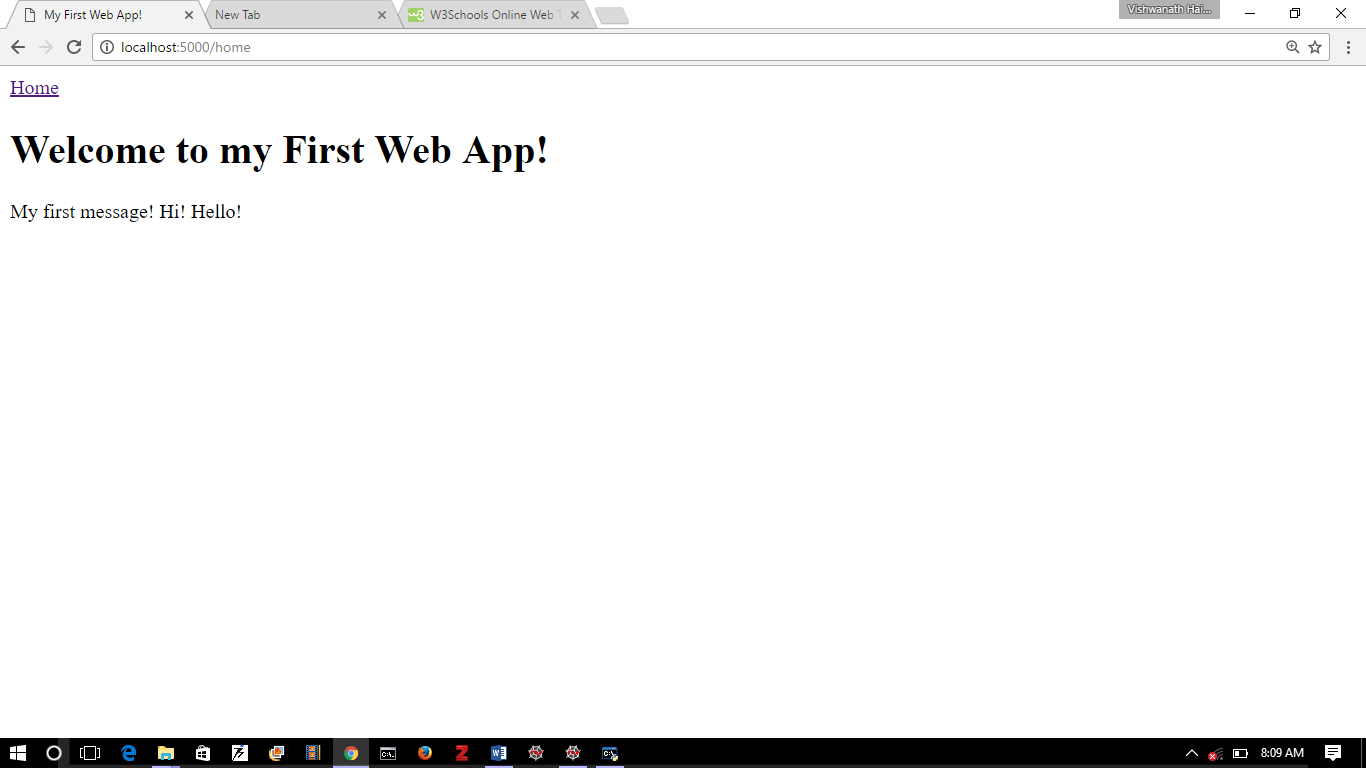
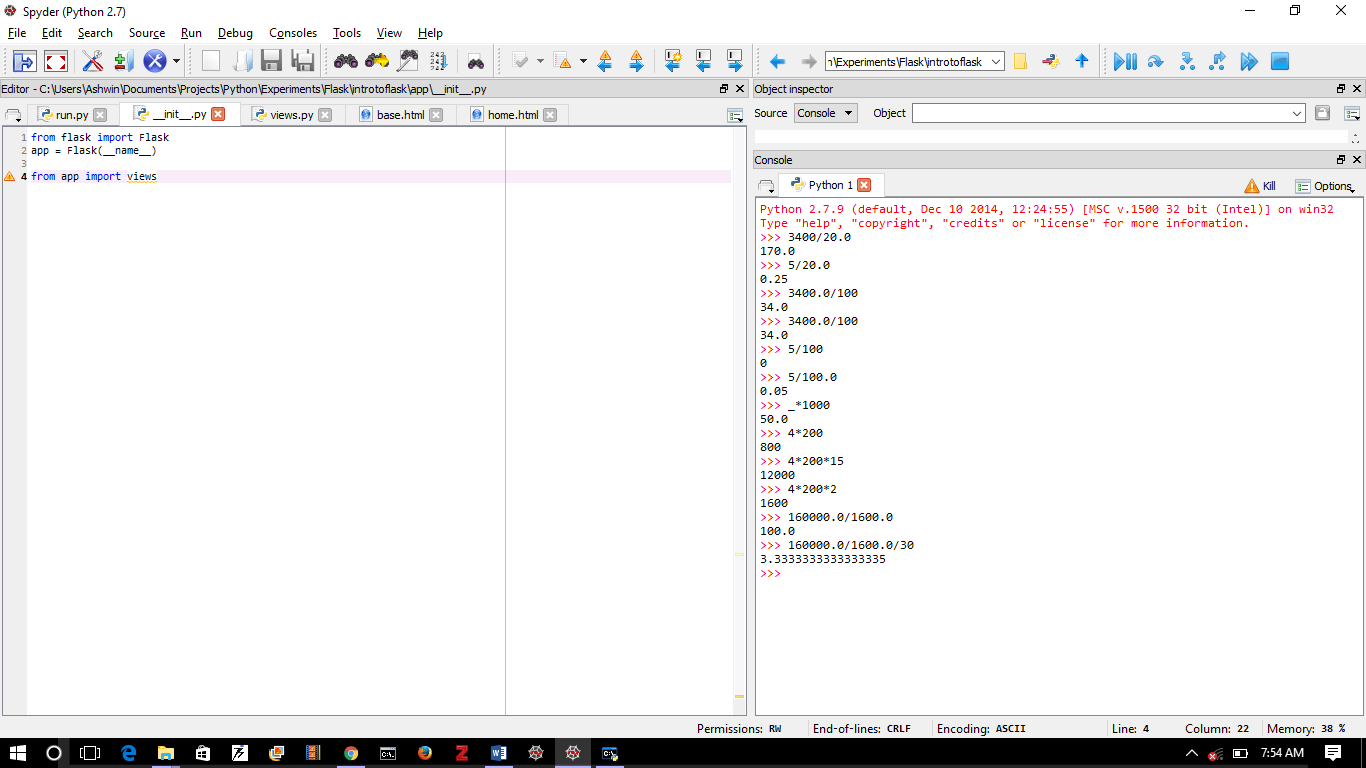
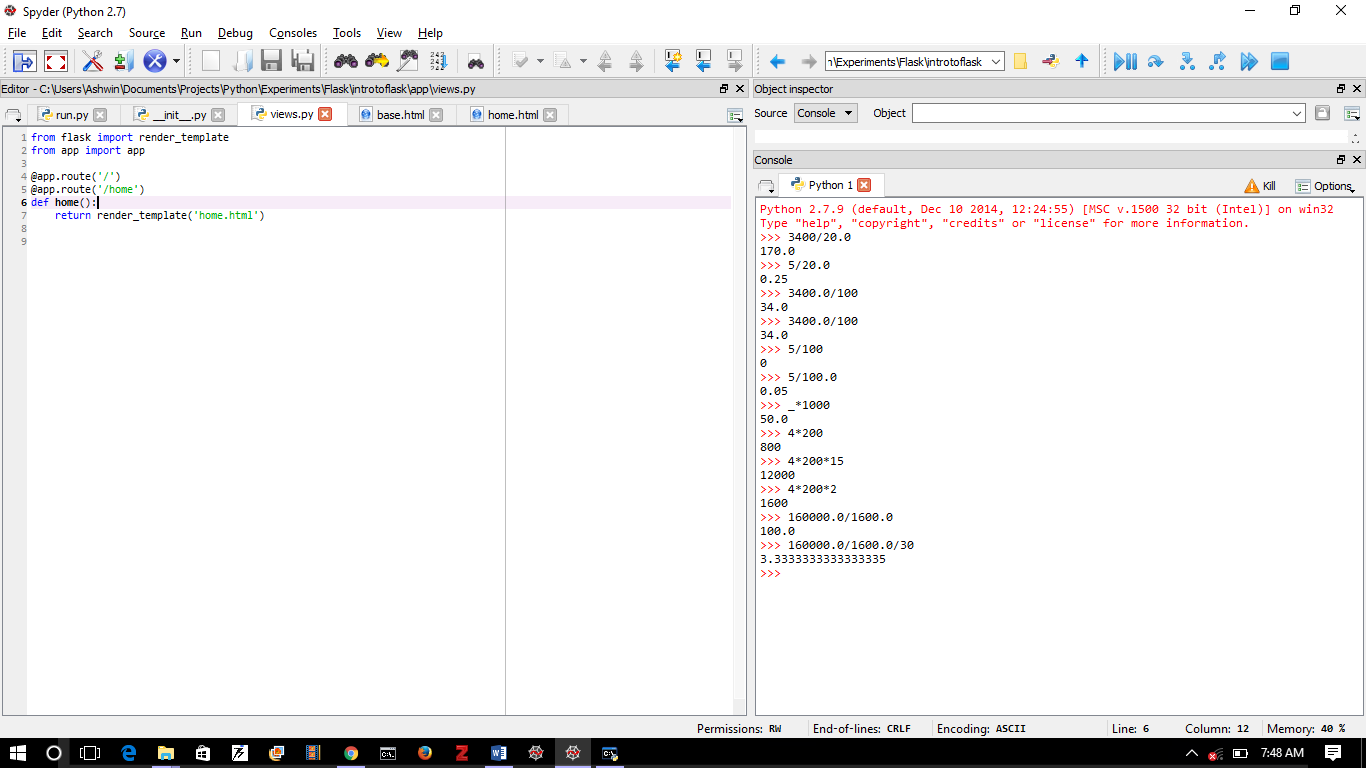
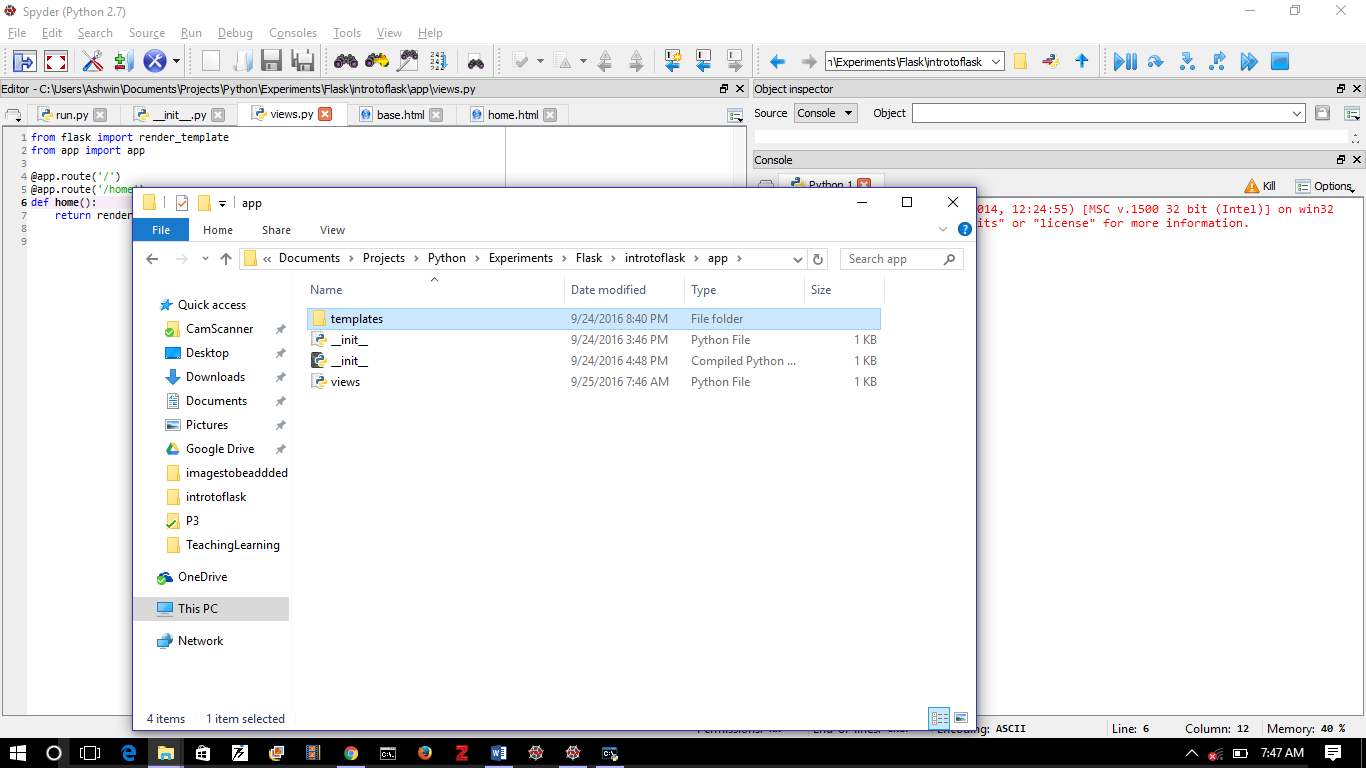


Figure 10: The “views.py” module that tells Flask how to connect the templates to the app. Notice the funny “@” symbols? They are called “decorators” and they modify Python functions. In the “views.py” module, we import the “render\_template” function (which simply takes a template and displays it). We also import “app”. We decorate the function home() with @app.route(“/”) and @app.route(“/home”). This means that whenever the base page or the home page are called, the function home is run. We also have to connect views.py to the app by adding a line in \_\_init\_\_.py to import views. If you go to the browser and just refresh the <http://localhost:5000/> page, you will see the home.html template rendered! You don’t have to restart the server ☺

# Slightly More Involved App

Well, we can’t do much with this. For one thing, the templates are all dead (i.e. static). They can’t really interact with the user. We haven’t given Python a role yet. Let’s do that now.

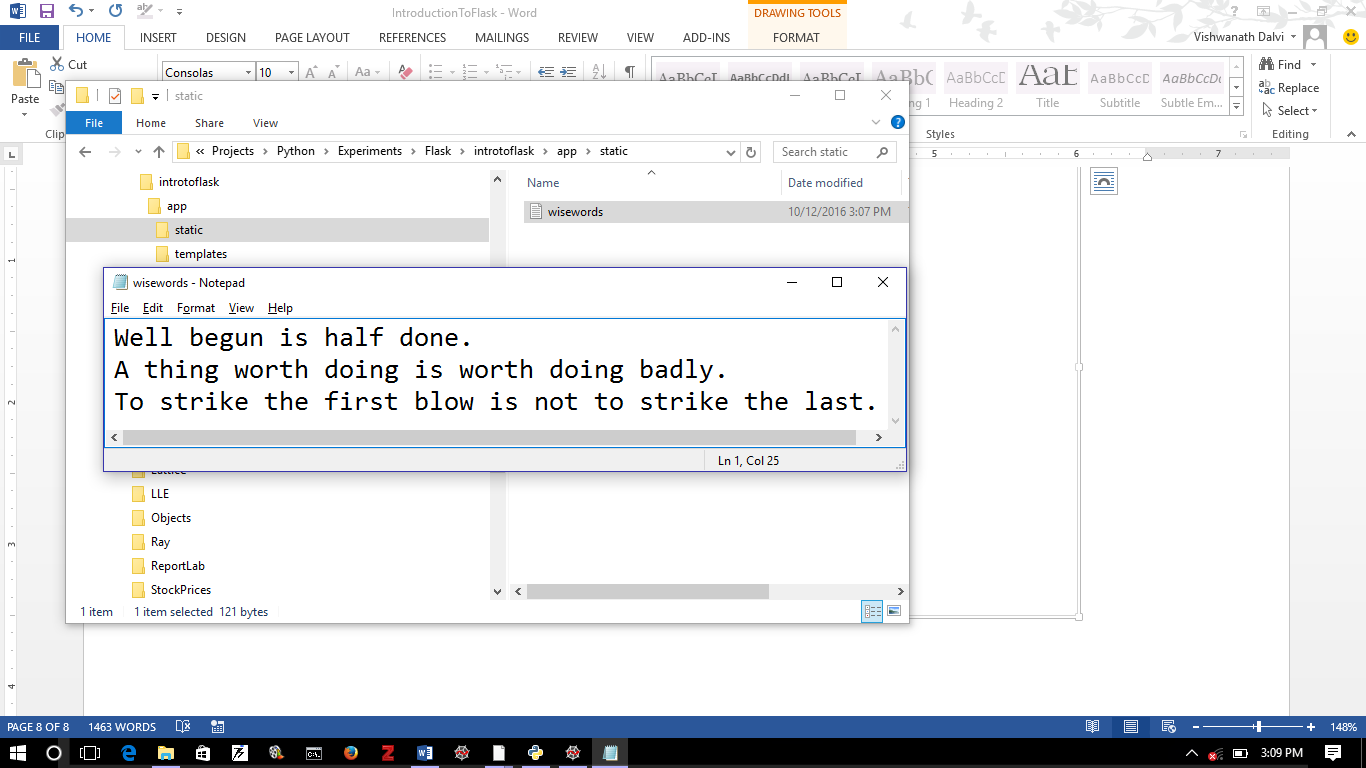
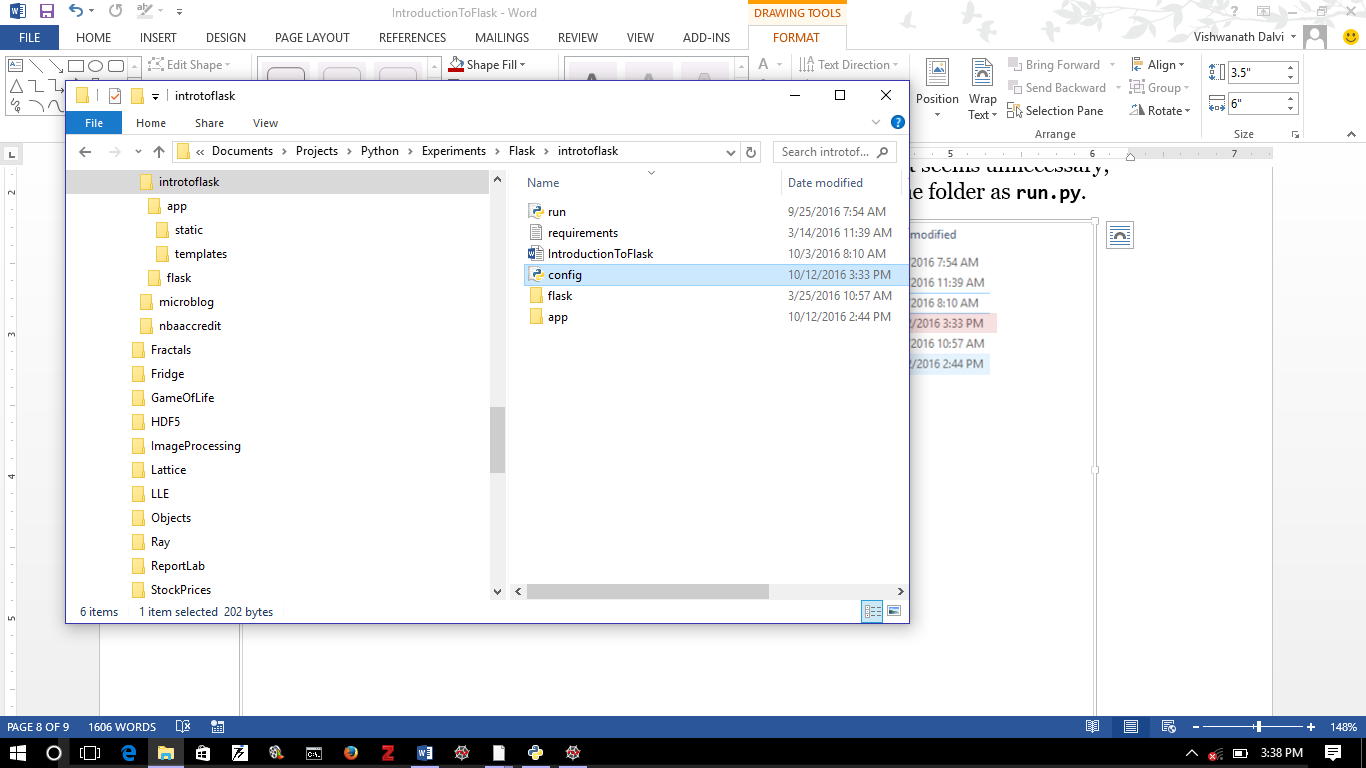


Figure 11: A text file called wisewords.txt is placed in the app\static folder.

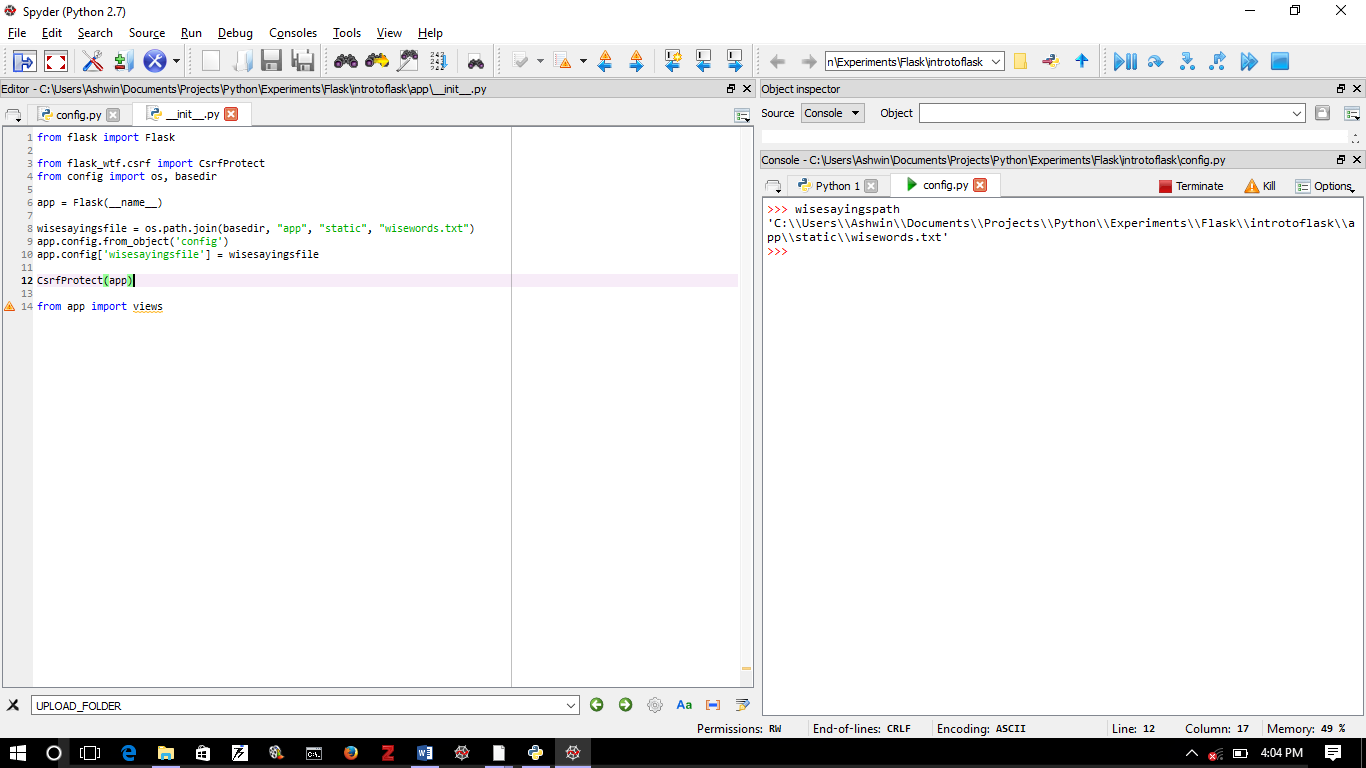
How about a thought for the day at the top of the page? That would be nice ☺. Let’s make a text file called wisewords.txt and put it in a special place: a folder called static that lives in the app folder. See Figure 11.

We have put three very wise sayings in it. The sayings themselves are well worth pondering over – and all are quite relevant to this exercise. But let’s move on …

We want one of these sayings to appear on top of each of our pages. But first we have to *read* the file. But to do that we have to *find* the file. And that is not so simple! You could just write a relative reference like /static/wisewords.txt but you will run into trouble as soon as you try to run your app on another filesystem! Let’s do something that seems unnecessary, but is good practice. Let’s make a config.py file. This lives in the same folder as run.py. See Figure 12.



**config.py**



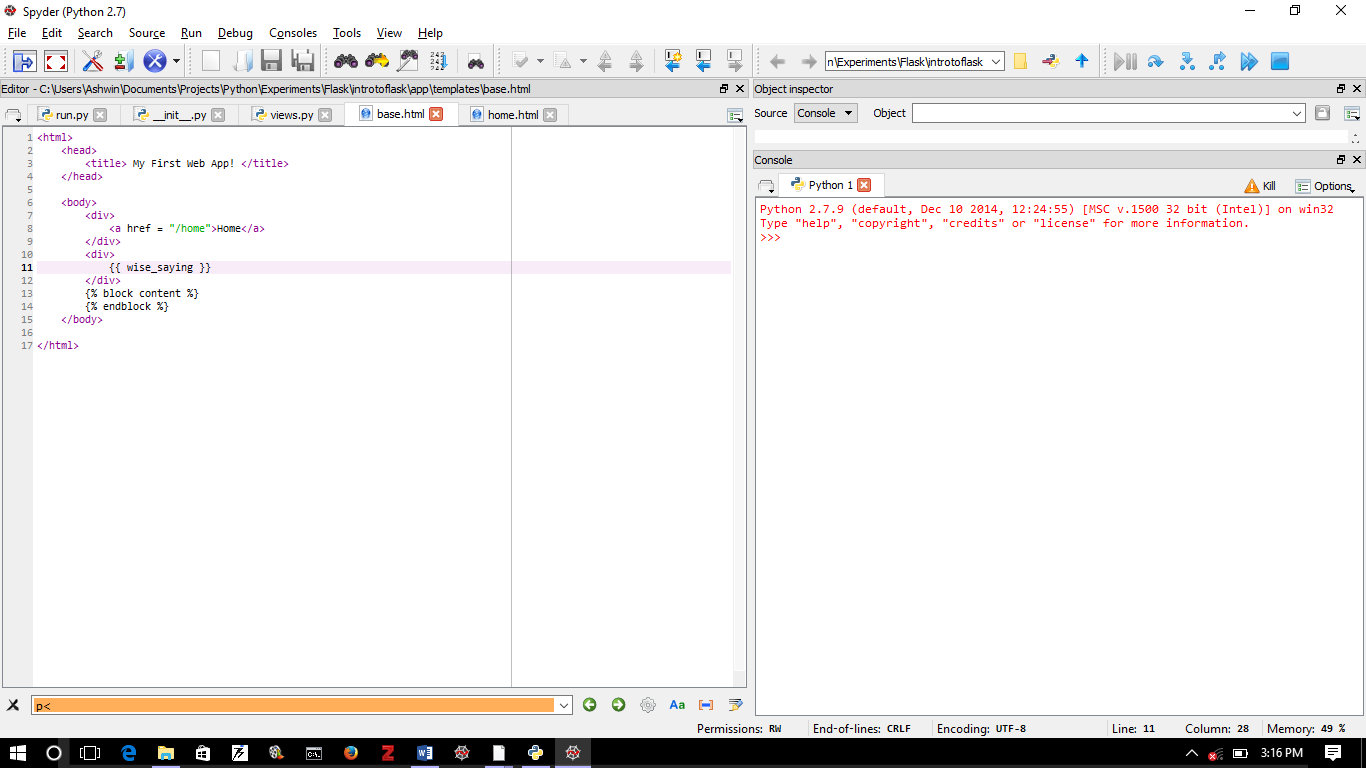
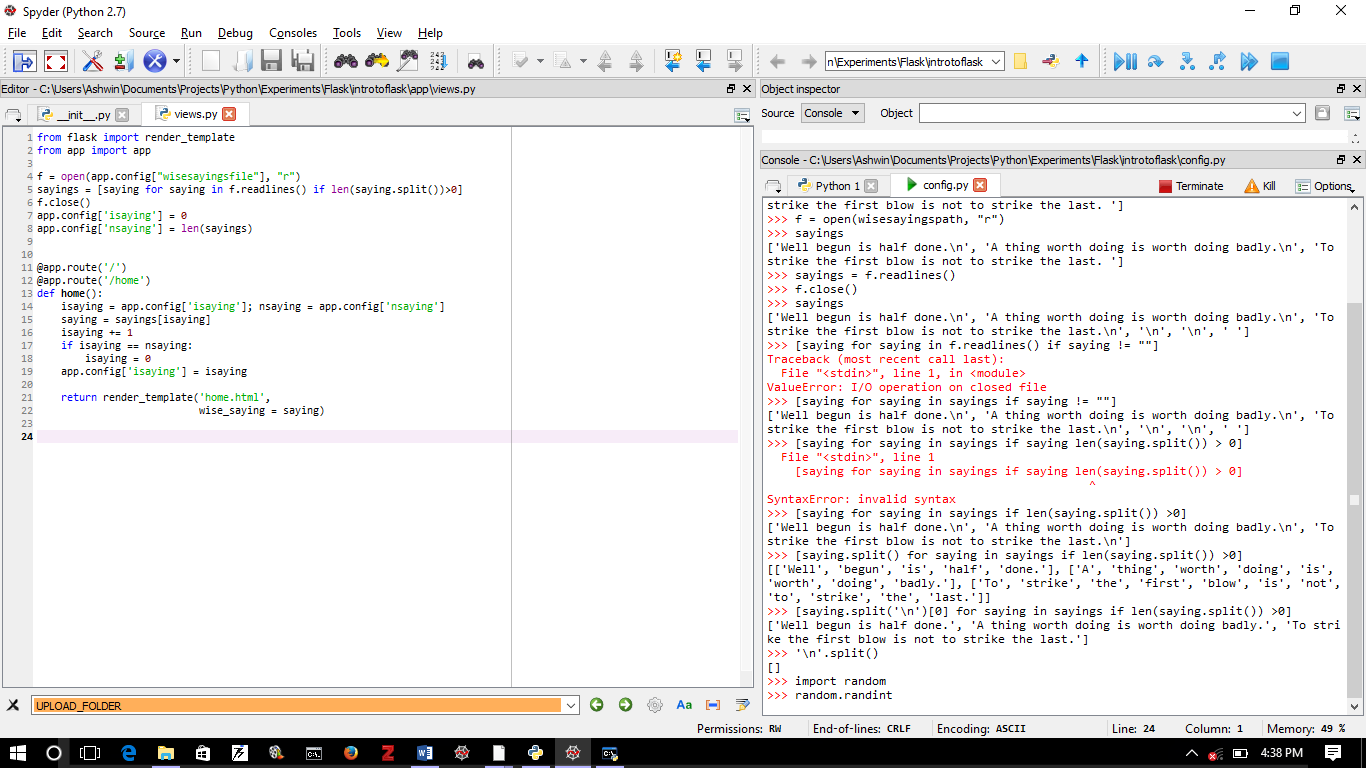
**\_\_init\_\_.py**

Figure 12: Making the config.py file. The WTF\_CSRF\_ENABLED is necessary as is the SECRET\_KEY. Don’t worry about it right now. Notice how the os.path.join command works. If you change the filesystem, the command will still work and return the path in that filesystem’s nomenclature! You need to make some small changes to the \_\_init\_\_.py file in the app folder. The changes you need to make are highlighted.

The purpose of the config.py file is, among other things, to hold parameters that are applicable to the whole app. The CsrfProtect module is necessary for security purposes. Just use it as indicated ☺.

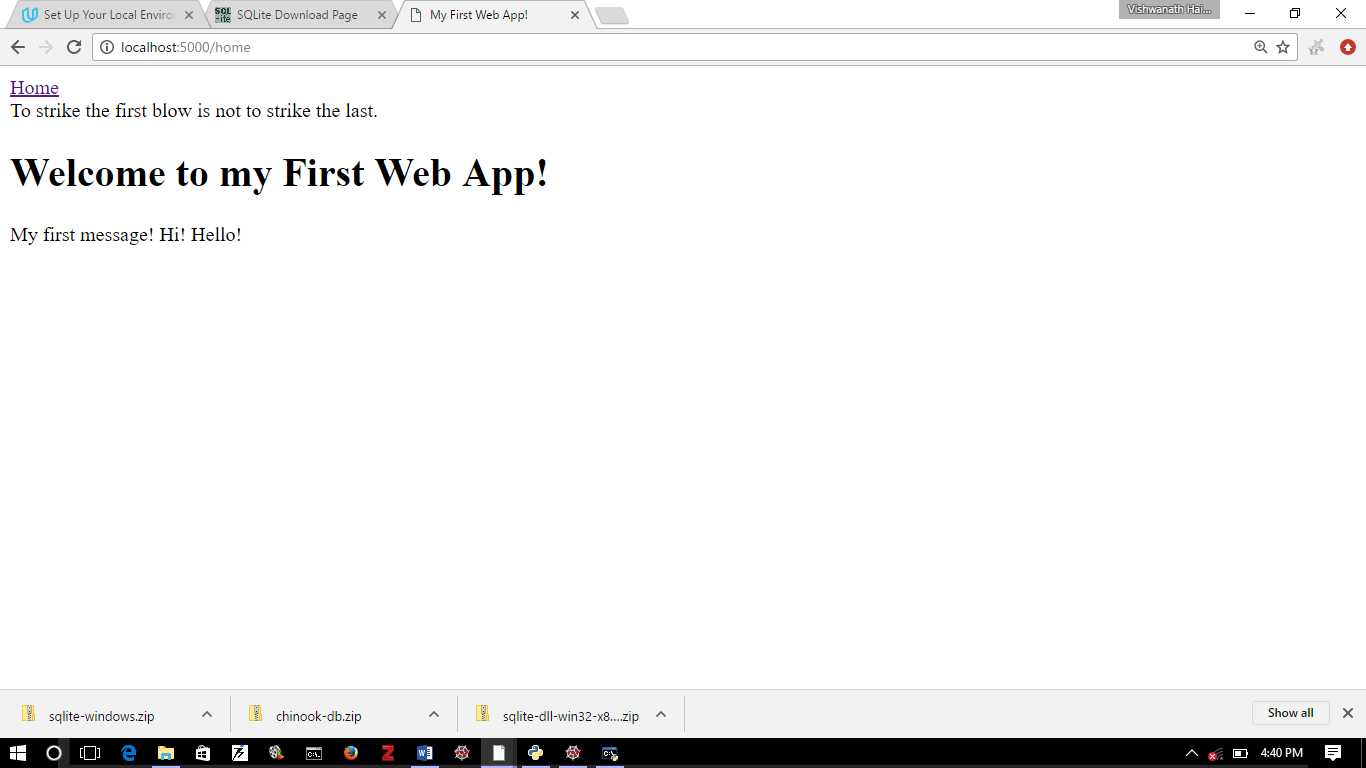
We haven’t really don’t anything yet: I just used this opportunity to introduce the config.py file. Let’s do something now …

First, let’s make a small change in the base.html template. Let’s make another change in the views.py file. See both changes, along with the result in the running app, implemented in Figure 13.



base.html

views.py



**The running app!**

Figure 13: Notice two changes (highlighted) in the base.html and views.py files. Most of the changes in the views.py file has to do with selecting a saying from our wisewords.txt file (Can you interpret the code? Notice how the app.config can be used as a dictionary). The important part is the addition to the render\_template function: we are passing a keyword argument. The keyword in this case is wise\_saying. Notice that the same keyword appears in the base.html inside the {{}}. We are using, behind the scenes, a library called jinja to replace the value in {{}} with the appropriate html. The result can be seen upon running the app. The saying will update every time you click the Home link!

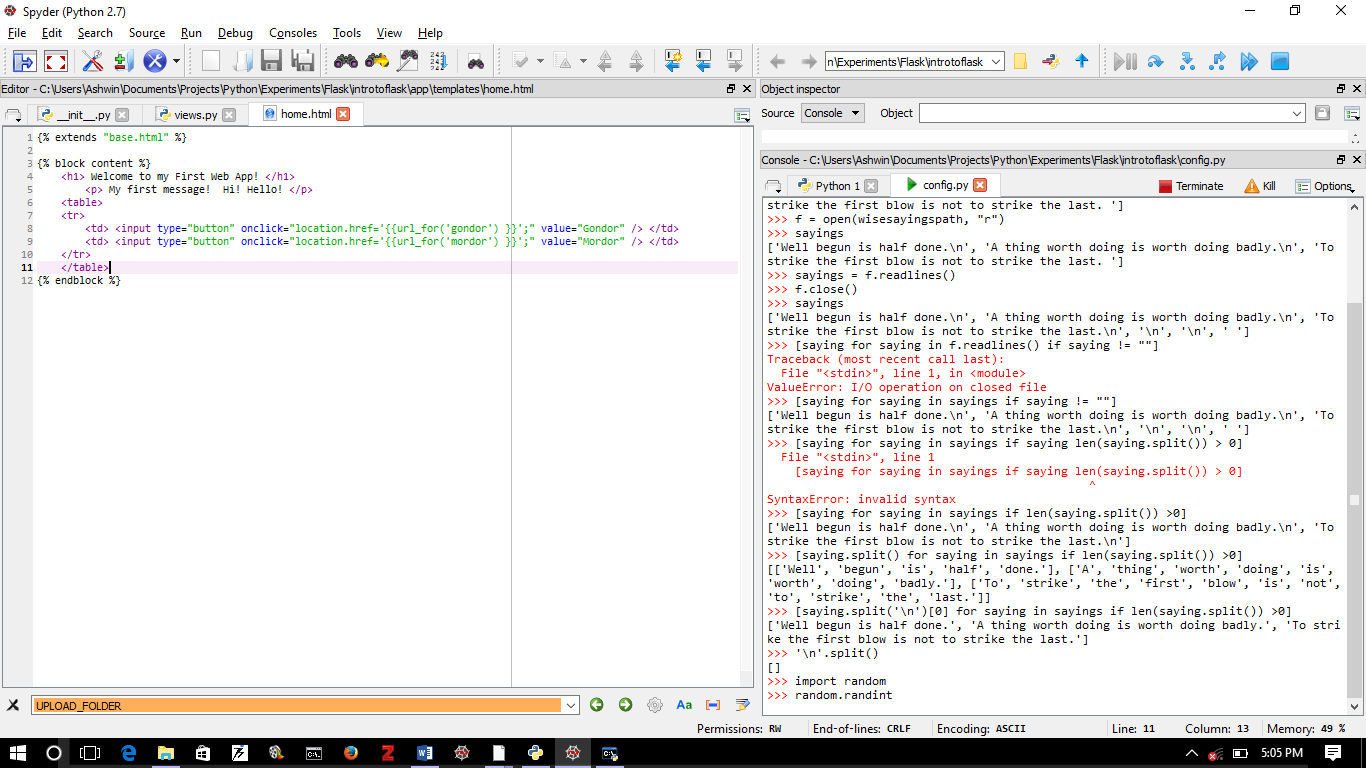
You have just had Python dynamically modify a webpage using a result of your code! How cool is that!

# Multiple Pages

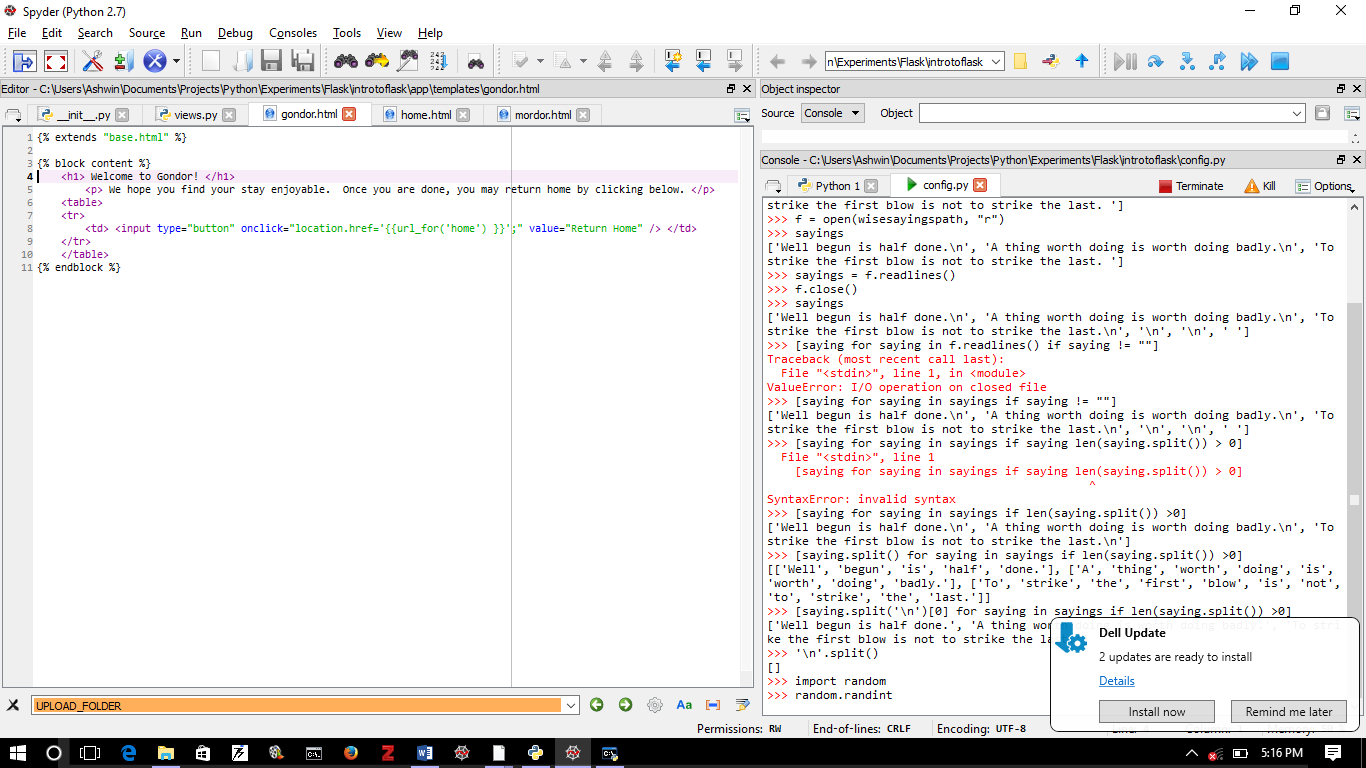
What fun is an app with a single page? Let’s make a couple of pages and put a link to them from the home page. Just for fun, we will use pushbuttons to create the links. So here we go. We are going to modify the home.html template to add buttons to the new pages. The new templates are gondor.html and mordor.html. We are going to write templates for the pages and tell views.py how to render them using the functions gondor and mordor respectively. All of this is, of course, done in Figure 14.

Note that in views.py, we have imported another module called url\_for. The reason for this is similar to the os.path.join thinking: to insulate us from vagaries of various operating systems and pages. The url\_for automatically adds the information for the host environment.

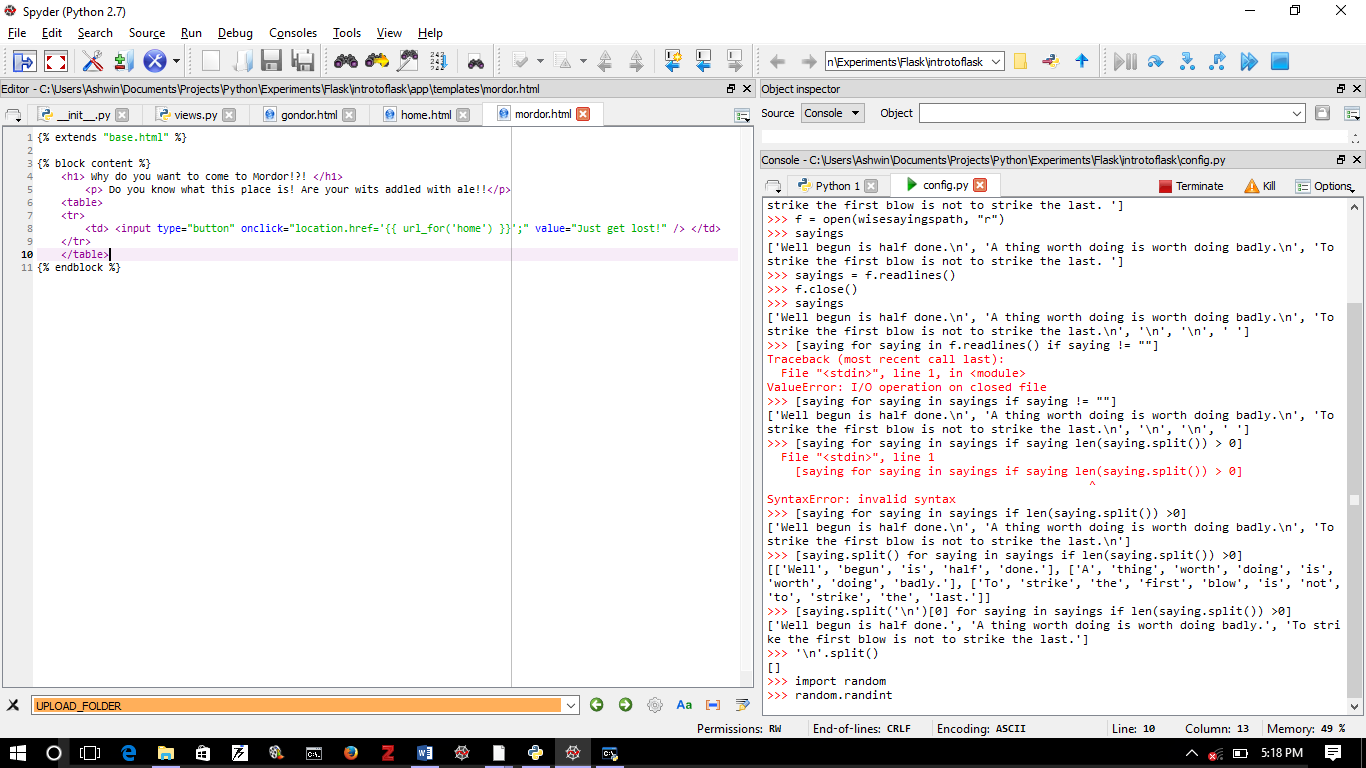
Now, for those of you who worry about these things, can you guess why the wise saying does not appear on the top of either the Gondor or the Mordor page? Keep reading and stay sharp!



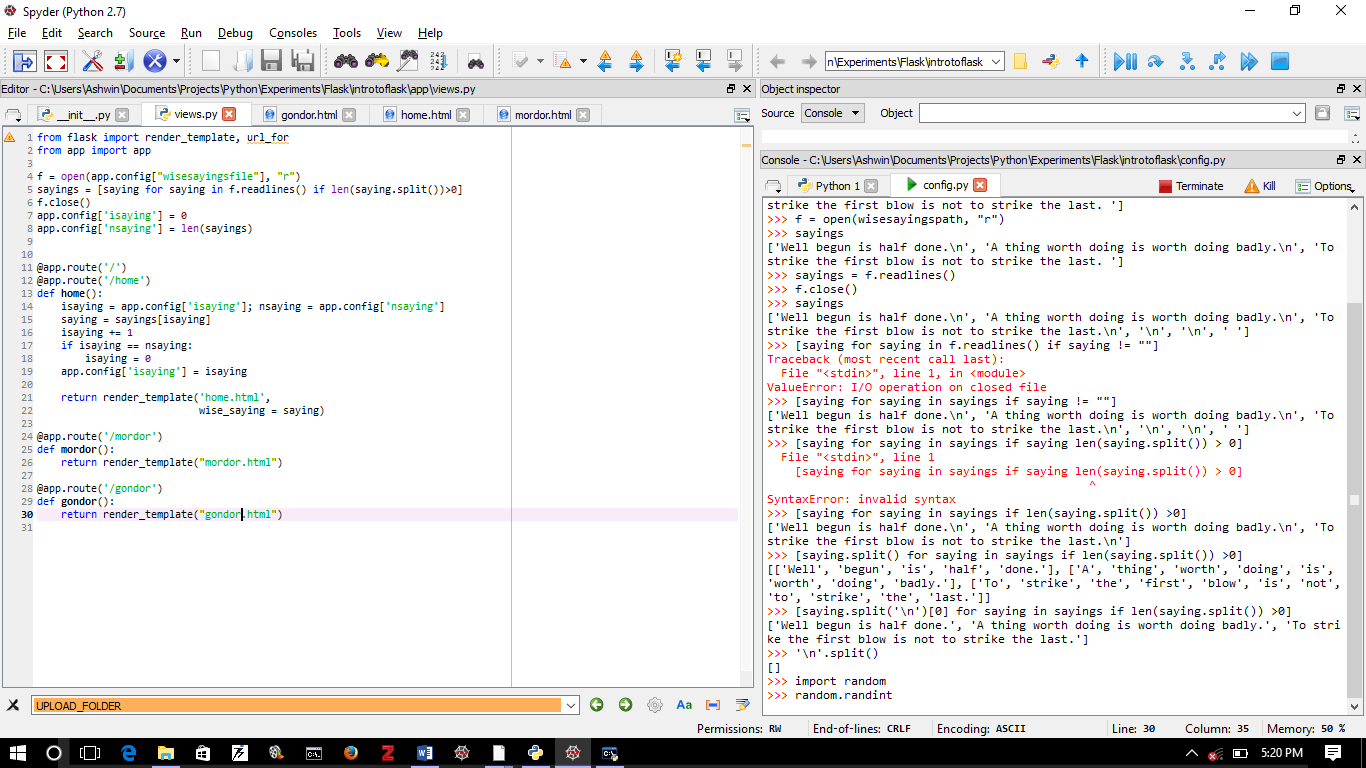
**home.html**



**gondor.html**



**mordor.html**

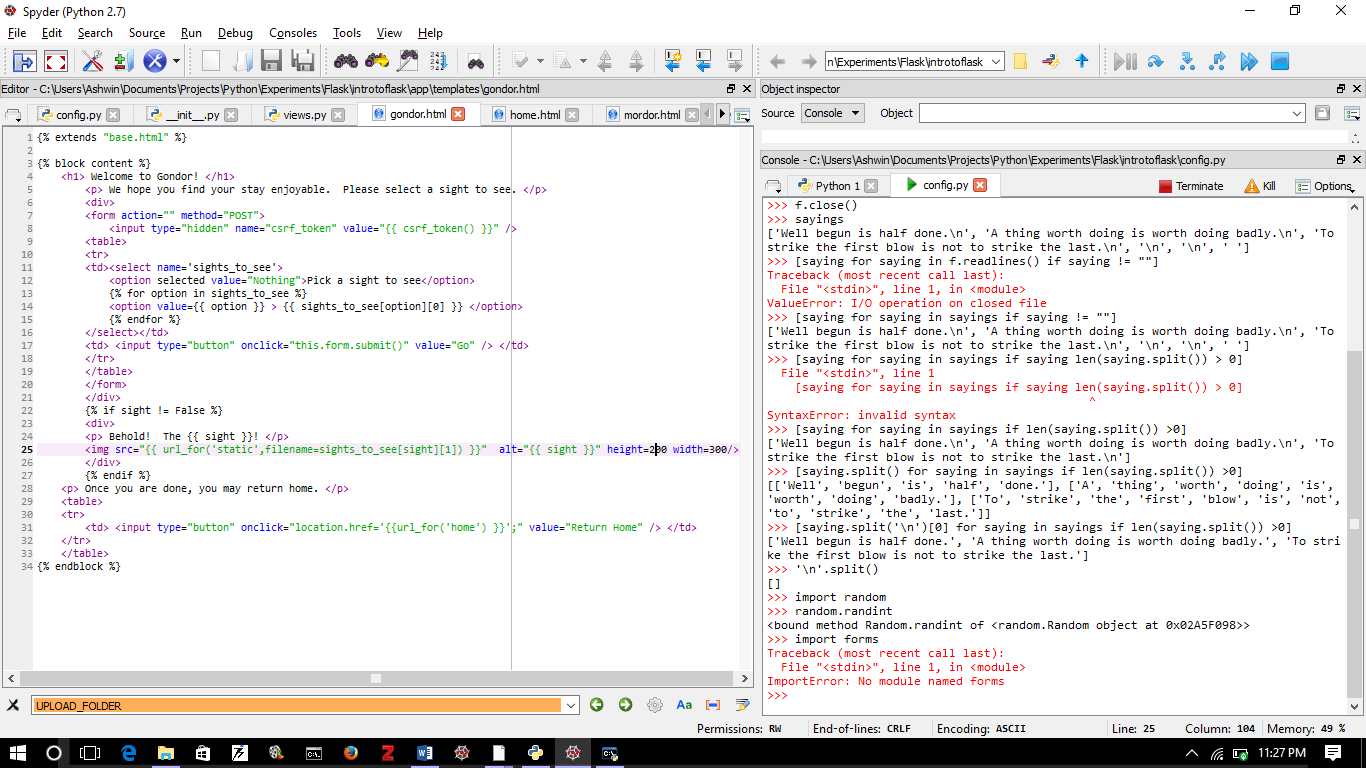


**views.py**

Figure 14: We have added two more pages to our web-app. The new pages have the templates gondor.html and mordor.html, both in the app\templates folder. How to bring them up is described in the view.py file under the corresponding decorators. Notice how we use the HTML widget for buttons. This is a great convenience since now we can tap into the entire HTML, CSS, Javascript ecosystem!

Alright, let’s give our users a tour of Gondor. We will download 3 .jpg images from the internet (all three for Gondor) and put them all in the app\static folder. The three images are “Minas\_Tirith.jpg”, “White\_Tree.jpg” and “Soldiers\_of\_Gondor.jpg”.

We need to modify gondor.html as follows:



Form

Essential bit of code

Note use of a for loop

Note use of an if block

**Notice a Submit Button**

Figure 15: Modifications to gondor.html to incorporate a drop-down menu form. The menu (in the select tags which is given the name sights\_to\_see) is put within form tags. The CSRF protection is essential. A submit button is provided. Once clicked, it generates a POST that can trigger action by the Python backend.

We now need to modify the views.py file as follows:

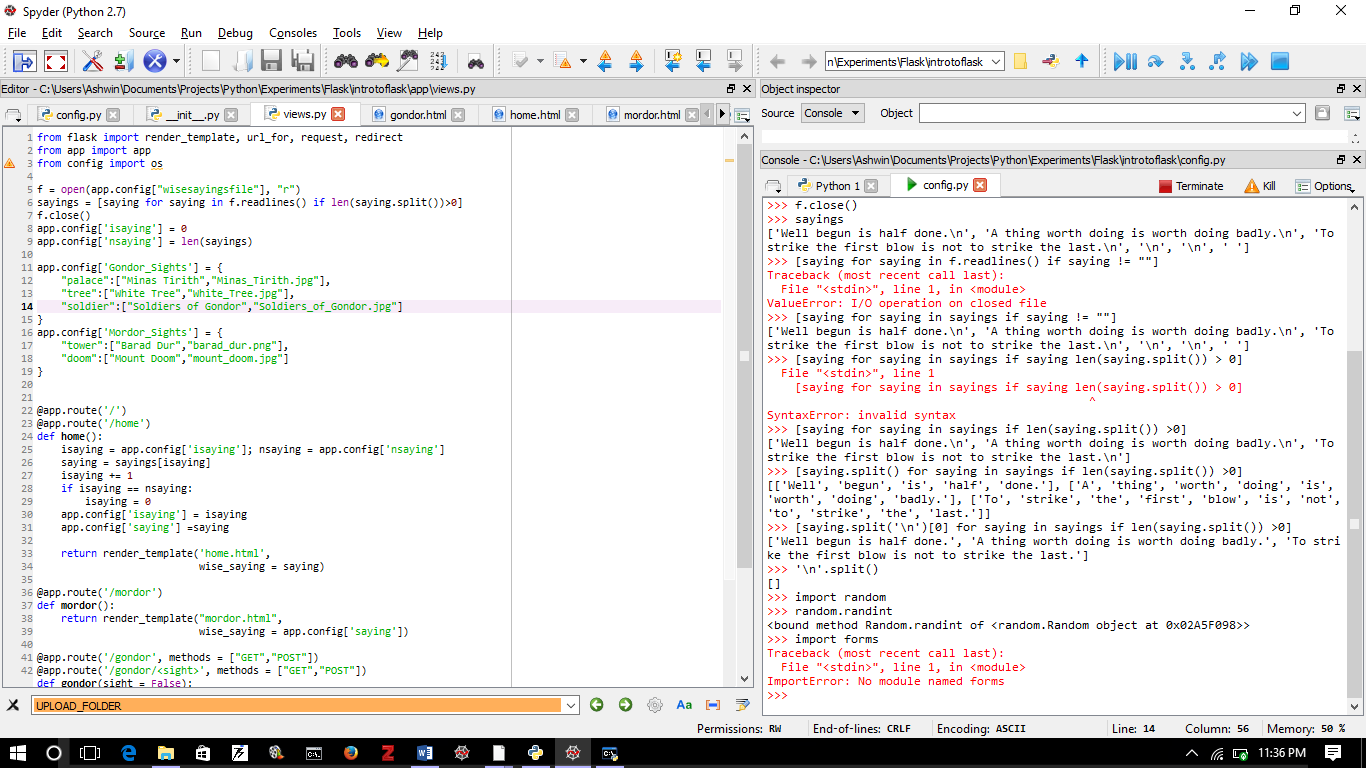


Figure 16: Part of the modified views.py file. We have imported new modules: request and redirect. We have also added a dictionary to app.config called Gondor\_Sights. The values of the dictionary are 2 element lists. The first element is a name of the sight, the second is the name of the file.

Finally we modify the gondor function as follows:

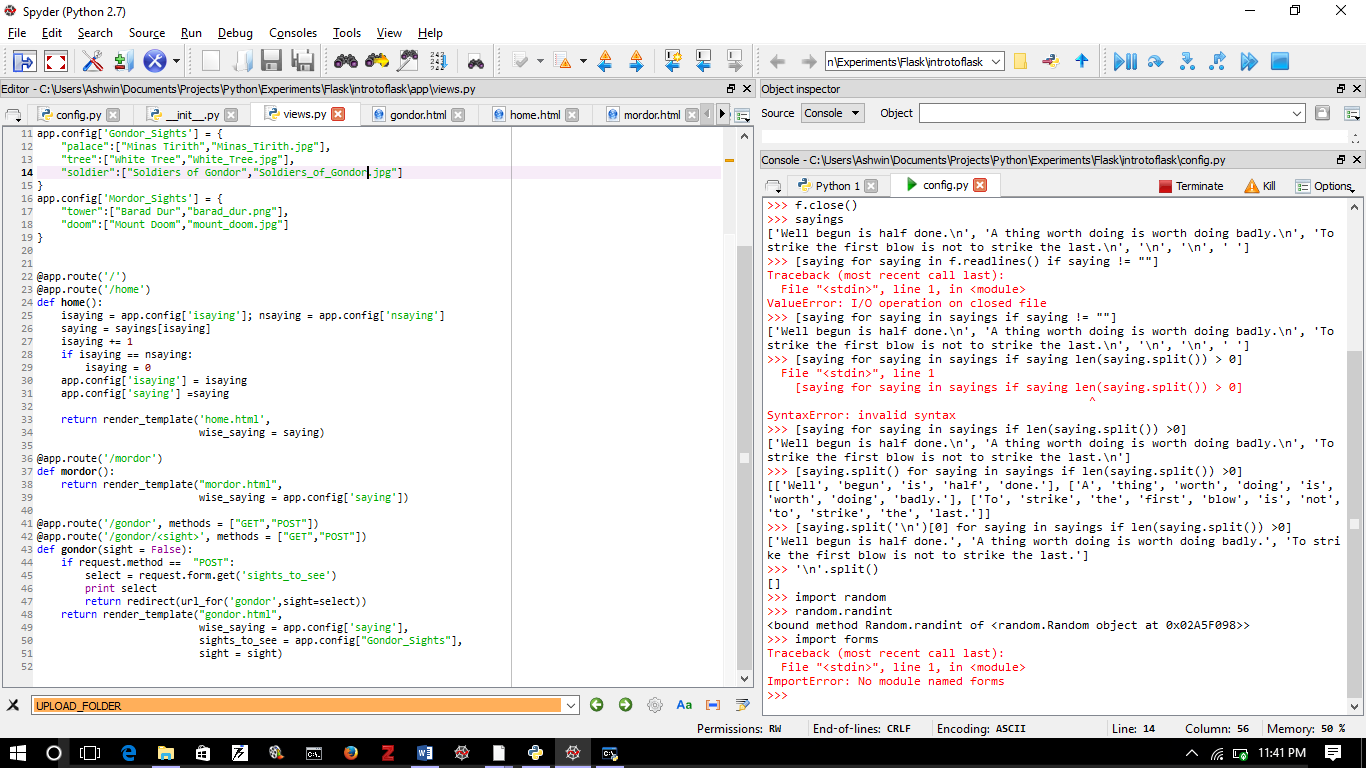


Figure 17: Modified gondor function. Notice that the decorators are modified and that there is an additional decorator. The gondor function now takes an argument called sight whose default value is False. There is an if block within the function that kicks in if a POST method is detected. The value selected from the drop-down menu is caught in the variable select which is obtained from the request.form.get(<name of menu>) .

You can use other HTML forms. See <http://www.w3schools.com/> for detailed tutorials on all things related to web-site development.

# Challenge

Make another web-page (template and a function in views.py) such that it takes two different numbers and displays their product. You have to use the internet!

1. Note the double “f”! ☺ [↑](#footnote-ref-1)
2. blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world [↑](#footnote-ref-2)
3. *All hail Miguel Grinberg!* [↑](#footnote-ref-3)