Table of Contents

[Django Setup 1](#_Toc88830149)

[Deactivating the Virtual Environment 3](#_Toc88830150)

[Stopping the Server 3](#_Toc88830151)

[Creating the First Django Application 3](#_Toc88830152)

[Adding View 5](#_Toc88830153)

[Creating a Base Template 7](#_Toc88830154)

[Sending Data to the View 8](#_Toc88830155)

[Posting Data to the Server 9](#_Toc88830156)

[Text Box Handling 12](#_Toc88830157)

[Training 13](#_Toc88830158)

[Production Code 16](#_Toc88830159)

## Django Introduction

Django is a full stack web development application framework where applications can be written using Python.

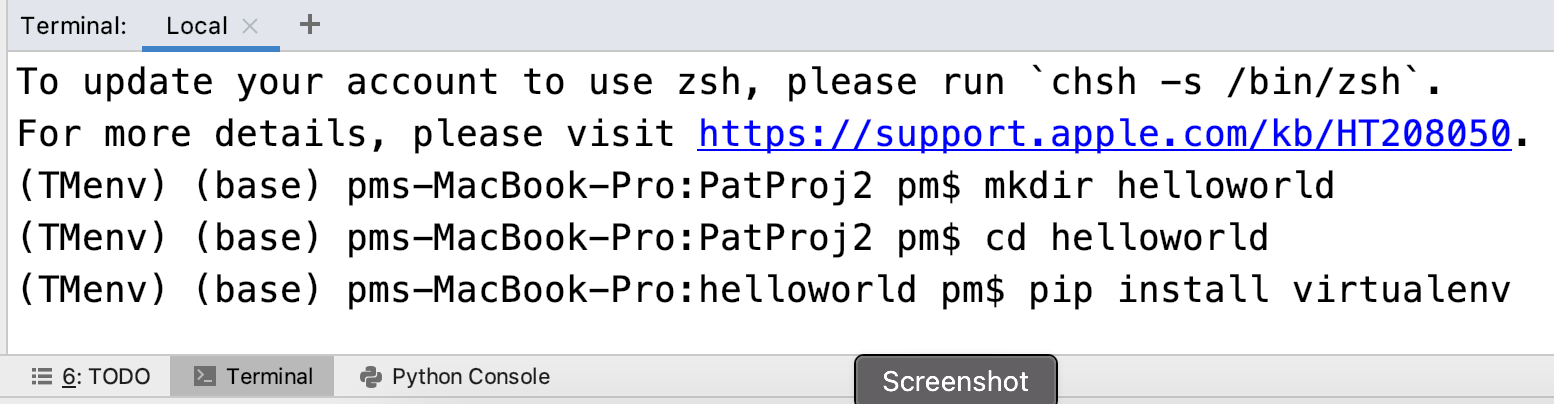
## Django Setup

These notes are based on documentation at:

<https://djangoforbeginners.com/hello-world/>

Example 1: Django Setup

This example will show how to create a Django application. I am going to keep it within my main PyCharm project folder. I will also use the PyCharm terminal.



These are the steps to start set up the Django project:

1. Create a directory:

mkdir helloworld

1. Move into the directory:

cd helloworld

1. Install the virtual environment:

pip install virtualenv

1. create a virtual environment.

virtualenv mypython

1. Activate the virtual environment:

On Mac

source mypython/bin/activate

On Windows (this is the path the Activate.ps1)

mypython\Scripts\Activate

1. Install the latest release of Django

pip install django

1. Create a a config directory:

(helloworld) $ django-admin startproject config .

The project directory will look like this.

(helloworld) $ tree

.

├── Pipfile

├── Pipfile.lock

├── config

│ ├── \_\_init\_\_.py

| ├── asgi.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

└── manage.py

1 directory, 8 files

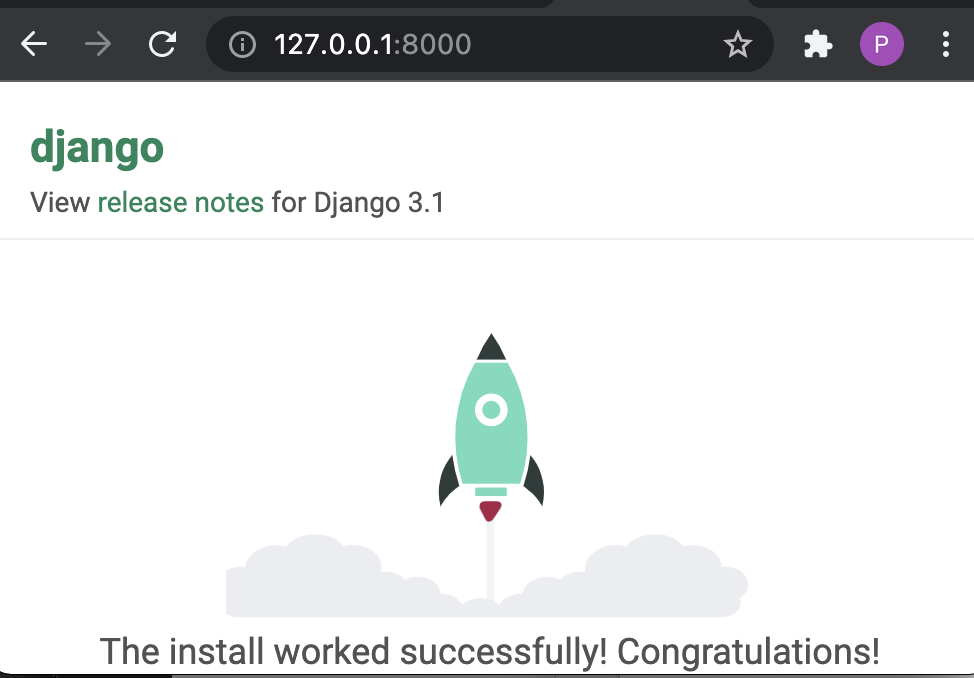
Here is a quick summary of each file:

* The **config/settings.py**file controls our project’s settings.
* **urls.py** tells Django which pages to build in response to a browser or URL request.
* **wsgi.py**, which stands for [Web Server Gateway Interface](https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface), helps Django serve our eventual web pages.
* **manage.py** file executes various Django commands such as running the local web server or creating a new app.
* **asgi.py** file, new to Django as of version 3.0 which allows for an optional [Asynchronous Server Gateway Interface](https://asgi.readthedocs.io/en/latest/specs/main.html) to be run.

1. Finally, start the server to launch your application.

(.venv) > python manage.py runserver

When finished, the following output will be visible.



1. You may receive a message indicating that migrations are needed. To run the migration, stop the server by pressing **ctrl+c** twice. Then, implement the migration in terminal by typing;

python manage.py migrate

### Deactivating the Virtual Environment

(To deactivate the environment run the following command)

deactivate

### Stopping the Server

To get rid of the warnings stop the server and perform the migration. Stop the server by typing:

ctrl + c

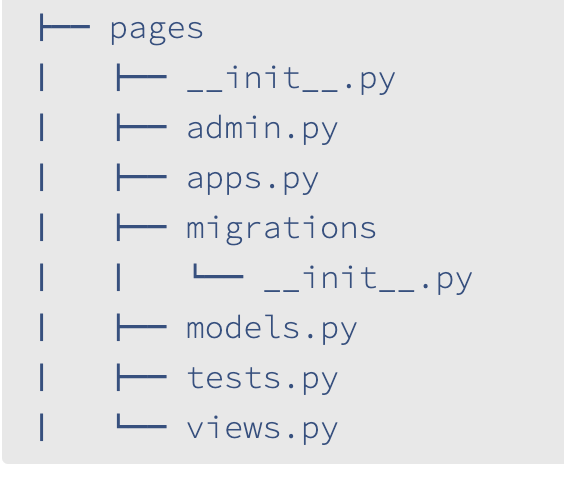
## Creating the First Django Application

Example 2: Creating the First Application

To create the pages application, type the following:

python manage.py startapp pages

Here is what the directory looks like:



Let’s review what each new pages app file does:

* admin.py is a configuration file for the built-in Django Admin app
* apps.py is a configuration file for the app itself
* migrations/ keeps track of any changes to our models.py file so our database and models.py stay in sync
* models.py is where we define our database models which Django automatically translates into database tables
* tests.py is for our app-specific tests
* views.py is where we handle the request/response logic for our web app

Even though our new app exists within the Django project, Django doesn’t “know” about it until we explicitly add it to the **config/settings.py** file. In your text editor open the file up and scroll down to INSTALLED\_APPS where you’ll see six built-in Django apps already there. Add pages.apps.PagesConfig at the bottom.

*# config/settings.py*

INSTALLED\_APPS **=** [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

"pages.apps.PagesConfig", *# new*

]

Let’s start by updating the views.py file in our pages app to look as follows:

# pages/views.py

from django.http import HttpResponse

def homePageView(request):

return HttpResponse('Hello, World!')

Then inside the parent directory add a urls file

touch pages/urls.py

Next add the following code to **pages/urls.py** to specify the view at the root url

# pages/urls.py

from django.urls import path

from .views import homePageView

urlpatterns = [

path('', homePageView, name='home')

]

Then add the following code to **config/urls.py**

# config/urls.py

from django.contrib import admin

from django.urls import path, include # new

from django.conf.urls import include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('pages.urls')), # new

]

Run the code now:

python manage.py runserver

## Adding View

Example 3: Adding View

In the pages folder, make a templates directory and put a home.html page in it.

(pages) $ mkdir templates

(pages) $ touch templates/home.html

In your HTML page put:

|  |
| --- |
| <b>Hello from home.html</b> |

In **config/settings.py** replace the existing DIRS definition with the following code:

'DIRS': [str(BASE\_DIR.joinpath('templates'))], # new



In **pages/views.py** replace the code with this:

|  |
| --- |
| # pages/views.py  from django.shortcuts import render, HttpResponseRedirect  from django.http import Http404  from django.urls import reverse  from django.views.generic import TemplateView  def homePageView(request):  return render(request, 'home.html') |

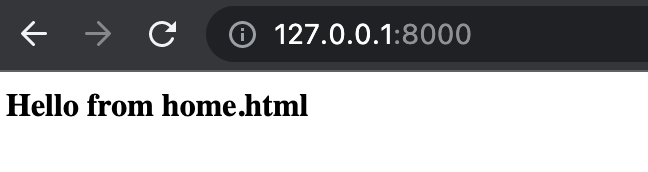
Then replace the code in **pages/urls.py** with the following. The items are highlighted:

|  |
| --- |
| # pages/urls.py  from django.urls import path  from .views import homePageView  urlpatterns = [  path('', homePageView, name='home'),  ] |

Then in the root directory run the following command:

python manage.py runserver

The content will appear when viewing the page.



Example 4: Adding an About Page

This example will walk you through the steps of adding another view.

**templates/about.html**

In the templates folder add an about.html page and place the following html in it:

|  |
| --- |
| <h3>About Page</h3> |

**pages/views.py**

Then in pages/views.py add a reference to the view.

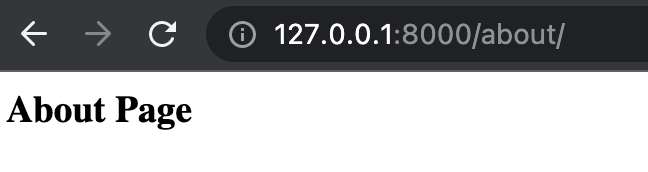
|  |
| --- |
| def aboutPageView(request):  # return request object and specify page.  return render(request, 'about.html') |

And finally import the view name and connect it to a URL at about/.

**pages/urls.py**

|  |
| --- |
| # pages/urls.py  from django.urls import path  from .views import homePageView, aboutPageView  urlpatterns = [  path('', homePageView, name='home'),  path('about/', aboutPageView, name='about')  ] |

Start up the web server with runserver and navigate to http://127.0.0.1:8000/about. The new About page is visible.



Exercise 1 (4 marks)

Add a view that uses yourfirstname.html and a path with your firstname. To get the marks you must use your actual name. Show your code in the following sections:

**templates/<firstname>.html**

|  |
| --- |
| <h3>Victor's Page</h3> |

Updated **pages/views.py** file.

|  |
| --- |
| from django.shortcuts import render  # Create your views here.   # pages/views.py from django.shortcuts import render, HttpResponseRedirect from django.http import Http404 from django.urls import reverse from django.views.generic import TemplateView  def homePageView(request):  return render(request, 'home.html')  def aboutPageView(request):  # return request object and specify page.  return render(request, 'about.html')  def victorPageView(request):  return render(request, 'victor.html') |

Updated **pages/urls.py** file.

|  |
| --- |
| # pages/urls.py from django.urls import path from .views import homePageView, aboutPageView, victorPageView  urlpatterns = [  path('', homePageView, name='home'),  path('about/', aboutPageView, name='about'),  path('victor/', victorPageView, name='victor') ] |

Show a screenshot of the view after running the server.

|  |
| --- |
|  |

## Creating a Base Template

For efficiency and styling, it is possible to implement a single base view that can be applied in all pages.

Example 5: Extending Templates

This example shows how to create a base view that displays shared content for the home and about pages. To create the base view add a base.html file to the templates folder and place the following code in it:

**templates/base.html**

|  |
| --- |
| <!-- templates/base.html -->  <a href="{% url 'home' %}">Home</a> |  <a href="{% url 'about' %}">About</a>  {% block content %} {% endblock content %} |

Next update the home page to include the base template while implementing custom content within the content block.

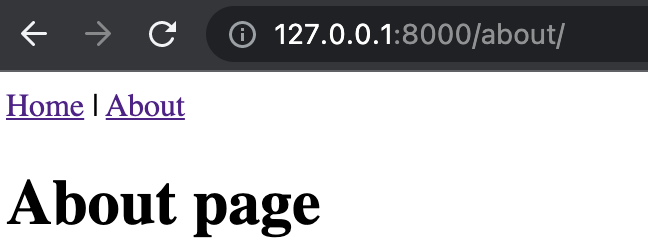
**templates/home.html**

|  |
| --- |
| <!-- templates/home.html -->  {% extends "base.html" %}  {% block content %}  <h1>Home page</h1>  {% endblock content %} |

And then update the **about.html** page to also implement the base template.

|  |
| --- |
| <!-- templates/about.html -->  {% extends "base.html" %}  {% block content %}  <h1>About page</h1>  {% endblock content %} |

After, the styling will be applied to all pages.



Exercise 2 (4 marks)

Add a hyperlink to your first name page that was created in Exercise 1. Also include the base template in your first name page. Show a screenshot of your updated first name page while viewing it in the browser here.

|  |
| --- |
|  |

## Sending Data to the View

Data is sent to the view through custom JSON objects. The content can then be parsed and displayed using templating script within the view.

Example 6: Sending Data to the View

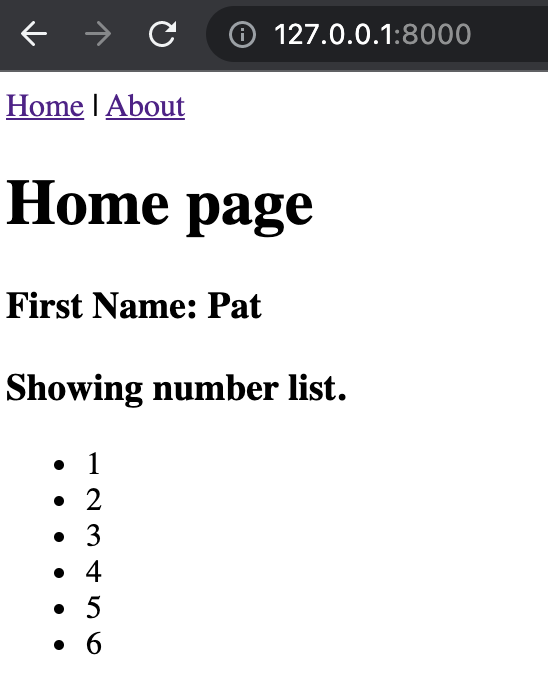
To add a custom data object to your response to the home view, add the highlighted code to your homePageView() function in **pages/views.py**.

|  |
| --- |
| def homePageView(request):  # return request object and specify page.  return render(request, 'home.html', {  'mynumbers':[1,2,3,4,5,6,],  'firstName': 'Pat'}) |

To display the custom content, add the following script to your **home.html** page:

|  |
| --- |
| <!-- templates/home.html -->  {% extends "base.html" %}  {% block content %}  <h1>Home page</h1>  <h3>First Name: {{firstName}}</h3>  <h3>Showing number list.</h3>  <ul>  {% for mynumber in mynumbers %}  <li>{{mynumber}}</li>  {% endfor %}  </ul>    {% endblock content %} |

When running the project, the content will display.



Exercise 3 (3 marks)

Modify the code in Example 6 so it displays your name instead of mine. Also send your last name to the view and show it beside your first name.

Show your updated **homePageView()** function here:

|  |
| --- |
| def homePageView(request):  # return request object and specify page.  return render(request, 'home.html', {  'mynumbers':[1,2,3,4,5,6,],  'firstName': 'Victor',  'lastName': 'Li'  }) |

Show your updated **templates/home.html** file here:

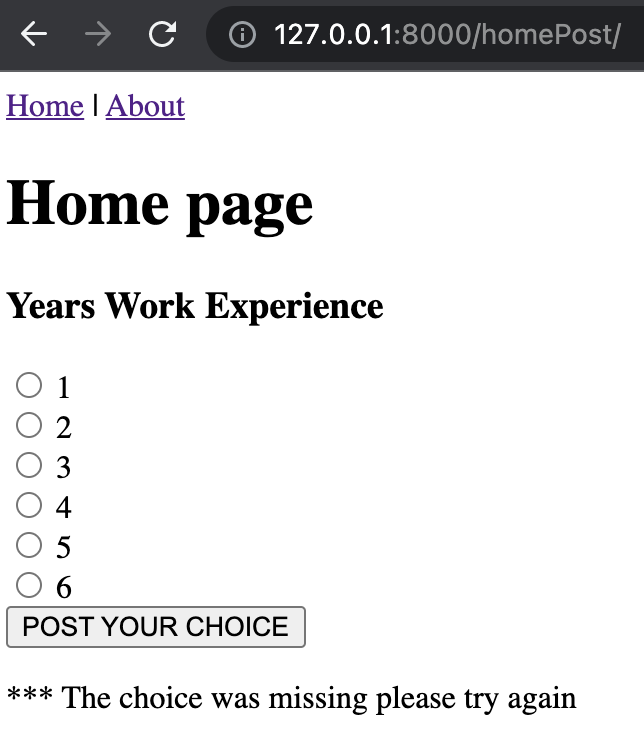
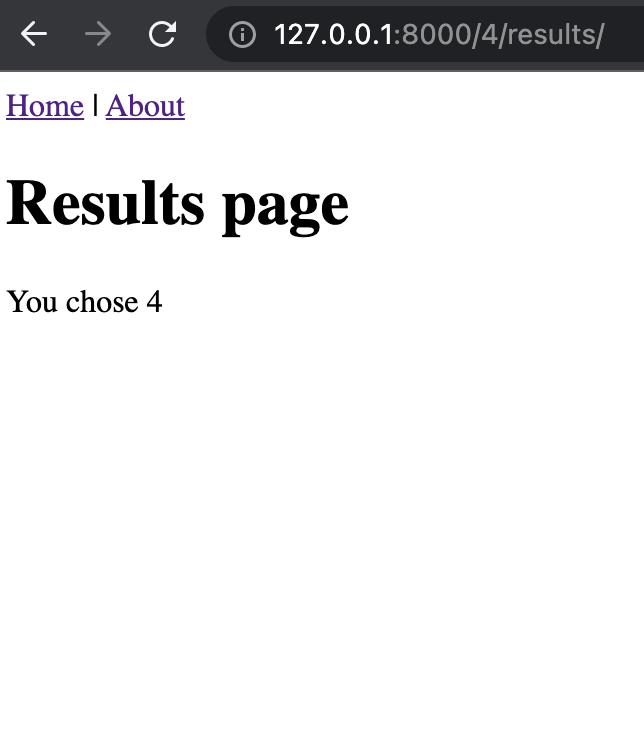
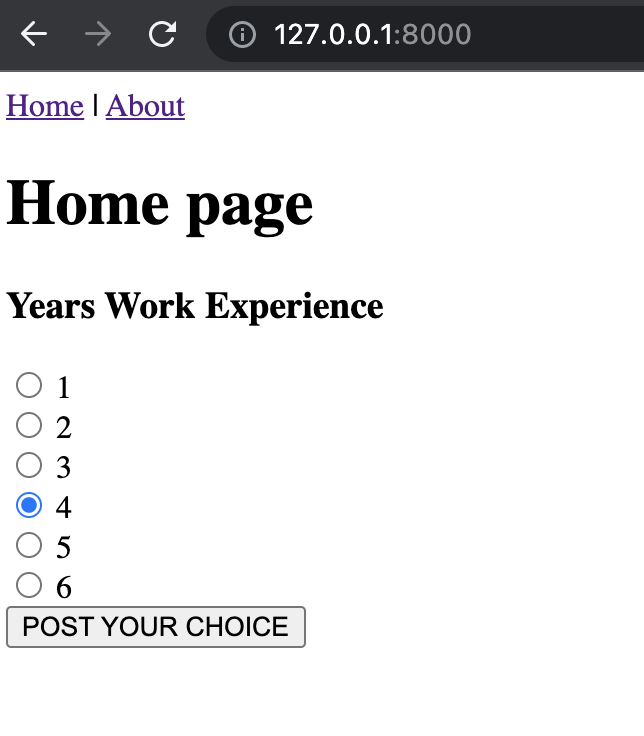
|  |
| --- |
| <!-- templates/home.html --> {% extends "base.html" %}  {% block content %}  <h1>Home page</h1>   <h3>First Name: {{firstName}}</h3>  <h3>Last Name: {{lastName}}</h3>    <h3>Showing number list.</h3>  <ul>  {% for mynumber in mynumbers %}  <li>{{mynumber}}</li>  {% endfor %}  </ul>  {% endblock content %} |

## Posting Data to the Server

When multiple data values are sent to the server during a web request, a POST request is often used. To post data to the server, a form is usually required. It also helps to have separate server methods to handle the request properly.

Example 7: Handling Post Requests

In this example you will enable post request handling for a radio button input. Values that are successfully posted will be displayed on a Results page through a redirect. When empty posts are made an error message will appear at the bottom of the page.



In the pages/views.py file add a new method to receive the post request and to enable the redirect to the results page.

**pages/views.py**

|  |
| --- |
| from django.http import HttpResponseRedirect  from django.urls import reverse  def homePost(request):  # Create variable to store choice that is recognized through entire function.  choice = -999  try:  # Extract value from request object by control name.  currentChoice = request.POST['choice']    # Crude debugging effort.  print("\*\*\* Years work experience: " + str(currentChoice))  choice = int(currentChoice)  # Enters 'except' block if integer cannot be created.  except:  return render(request, 'home.html', {  'errorMessage':'\*\*\* The choice was missing please try again',  'mynumbers': [1, 2, 3, 4, 5, 6, ]})  else:  # Always return an HttpResponseRedirect after successfully dealing  # with POST data. This prevents data from being posted twice if a  # user hits the Back button.  return HttpResponseRedirect(reverse('results', args=(choice,)))  def results(request, choice):  print("\*\*\* Inside reults()")  return render(request, 'results.html', {'choice': choice}) |

After the results and homePost functions are created, add the highlighted routes to pages/urls.py.

**pages/urls.py**

|  |
| --- |
| # pages/urls.py  from django.urls import path  from .views import homePageView, aboutPageView, results, homePost  urlpatterns = [  path('', homePageView, name='home'),  path('about/', aboutPageView, name='about'),  path('homePost/', homePost, name='homePost'),  path('<int:choice>/results/', results, name='results'),  ] |

Content that is posted to the server must usually be in a form. Note that the route and POST method are identified in the form header. The submit button is also contained in the form. As well, a conditional error display has been included to show an error message if one is defined.

**Templates/home.html**

|  |
| --- |
| <!-- templates/home.html -->  {% extends "base.html" %}  {% block content %}  <h1>Home page</h1>  <!-- This form references the homePost route. -->  <form action="{% url 'homePost' %}" method="post">  {% csrf\_token %}  <h3>Years Work Experience</h3>  <!-- Show radio button list named 'choice' -->  {% for choice in mynumbers %}  <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice }}">  <label for="choice{{ forloop.counter }}">{{ choice }}</label><br>  {% endfor %}  <!—Submit button is also contained in the form -->  <input type="submit" value="POST YOUR CHOICE">  </form>  <!-- End of form. -->  <!-- This error will notify the user if there are issues with the POST request -->  {% if errorMessage %}<p><strong>{{errorMessage}}</strong></p>{% endif %}  {% endblock content %} |

To display the results of the selection, add the following template.

**templates/results.html**

|  |
| --- |
| <!-- templates/home.html -->  {% extends "base.html" %}  {% block content %}  <h1>Results page</h1>  You chose {{choice}}  {% endblock content %} |

## Text Box Handling

Example 8: Handling an Additional Input

This example adds a text box control and handling to the solution for Example 7. To build this example, place the following code above the button tag inside home.html.

**templates/home.html**

|  |
| --- |
| <h3>GMAT Score</h3>  <input type="text" id="gmat" name="gmat"><br><br> |

Next, replace the homePost() and results() functions inside the view.py file.

**pages/views.py**

|  |
| --- |
| def homePost(request):  # Use request object to extract choice.  choice = -999  gmat = -999  try:  # Extract value from request object by control name.  currentChoice = request.POST['choice']  gmatStr = request.POST['gmat']  # Crude debugging effort.  print("\*\*\* Years work experience: " + str(currentChoice))  choice = int(currentChoice)  gmat = float(gmatStr)  # Enters 'except' block if integer cannot be created.  except:  return render(request, 'home.html', {  'errorMessage':'\*\*\* The data submitted is invalid. Please try again.',  'mynumbers': [1, 2, 3, 4, 5, 6, ]})  else:  # Always return an HttpResponseRedirect after successfully dealing  # with POST data. This prevents data from being posted twice if a  # user hits the Back button.  return HttpResponseRedirect(reverse('results', kwargs={'choice':choice,'gmat':gmat},))    def results(request, choice, gmat):  print("\*\*\* Inside reults()")  return render(request, 'results.html', {'choice': choice, 'gmat':gmat}) |

An adjustment to the routing is required to enable the results function to receive two parameters.

**pages/urls.py**

|  |
| --- |
| path('results/<int:choice>/<str:gmat>/', results, name='results'), |

Finally, the results.html view must be updated to also show the gmat score.

**templates/results.html**

|  |
| --- |
| <!-- templates/results.html -->  {% extends "base.html" %}  {% block content %}  <h1>Results page</h1>  You entered the following information:<br/>  Years work exerience {{choice}}<br/>  GMAT Score {{gmat}}  {% endblock content %} |

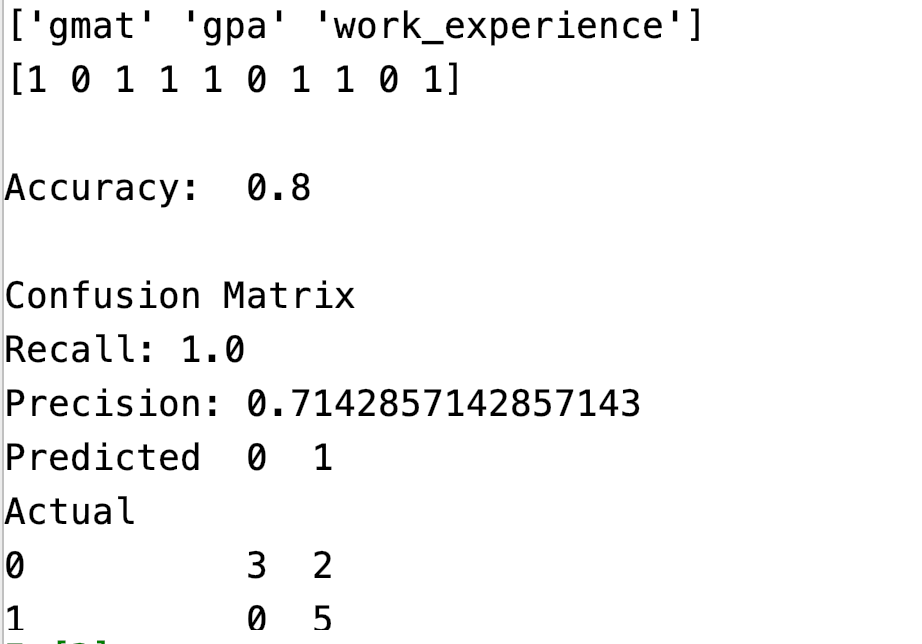
## Training

Now that our web application is set, we will want to use it later to handle our production code. First though we must train the model.

Example 9: Training a Logistic Regression Model

This example trains a logistic regression model to predict if a student is admitted to college. The model is saved as a pickle file. To ensure the model is saved properly it is loaded from the binary file and is then used to make predictions.

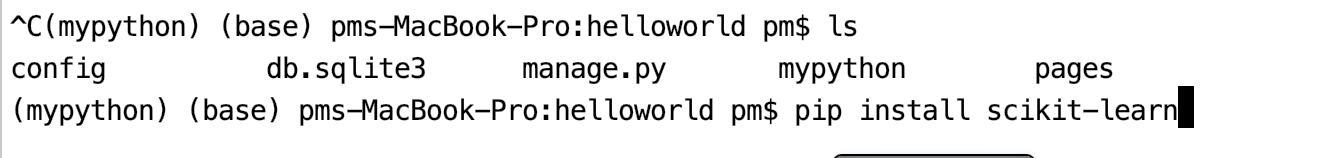
The results suggest that the model worked properly.



|  |
| --- |
| import pandas as pd  import numpy as np  import pickle  import sklearn.metrics as metrics  # Setup data.  candidates = {'gmat': [780,750,690,710,680,730,690,720,  740,690,610,690,710,680,770,610,580,650,540,590,620,  600,550,550,570,670,660,580,650,660,640,620,660,660,  680,650,670,580,590,690],  'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,  3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,  3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,  3.3,3.3,2.3,2.7,3.3,1.7,3.7],  'work\_experience': [3,4,3,5,4,6,1,4,5,  1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,  5,1,2,1,4,5],  'admitted': [1,1,1,1,1,1,0,1,1,0,0,1,  1,1,1,0,0,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,  0,0,1]}  df = pd.DataFrame(candidates,columns= ['gmat', 'gpa',  'work\_experience','admitted'])  print(df)  # Separate into x and y values.  predictorVariables = ['gmat', 'gpa','work\_experience']  X = df[predictorVariables]  y = df['admitted']  # Import the necessary libraries first  from sklearn.feature\_selection import SelectKBest  from sklearn.feature\_selection import chi2  # You imported the libraries to run the experiments. Now, let's see it in action.  # Show chi-square scores for each feature.  # There is 1 degree freedom since 1 predictor during feature evaluation.  # Generally, >=3.8 is good)  test = SelectKBest(score\_func=chi2, k=3)  chiScores = test.fit(X, y) # Summarize scores  np.set\_printoptions(precision=3)  print("\nPredictor variables: " + str(predictorVariables))  print("Predictor Chi-Square Scores: " + str(chiScores.scores\_))  # Another technique for showing the most statistically  # significant variables involves the get\_support() function.  cols = chiScores.get\_support(indices=True)  print(cols)  features = X.columns[cols]  print(np.array(features))  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  # Re-assign X with significant columns only after chi-square test.  X = df[['gmat', 'work\_experience']]  # Split data.  X\_train,X\_test,y\_train,y\_test = train\_test\_split(X, y, test\_size=0.25,  random\_state=0)  # Build logistic regression model and make predictions.  logisticModel = LogisticRegression(fit\_intercept=True, solver='liblinear',  random\_state=0)  logisticModel.fit(X\_train,y\_train)  # Save the model.  with open('model\_pkl', 'wb') as files:  pickle.dump(logisticModel, files)  # load saved model  with open('model\_pkl' , 'rb') as f:  loadedModel = pickle.load(f)  y\_pred=loadedModel.predict(X\_test)  print(y\_pred)  # Show confusion matrix and accuracy scores.  from sklearn import metrics  cm = pd.crosstab(y\_test, y\_pred, rownames=['Actual'], colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix")  recall = metrics.recall\_score(y\_test, y\_pred)  print("Recall: " + str(recall))  precision = metrics.precision\_score(y\_test, y\_pred)  print("Precision: " + str(precision))  print(cm)  # Create a single prediction.  singleSampleDf = pd.DataFrame(columns=['gmat', 'work\_experience'])  gmat = 550  workExperience = 4  admissionsData = {'gmat':gmat, 'work\_experience':workExperience}  singleSampleDf = pd.concat([singleSampleDf,  pd.DataFrame.from\_records([admissionsData])])  singlePrediction = loadedModel.predict(singleSampleDf)  print("Single prediction: " + str(singlePrediction)) |

## Production Code

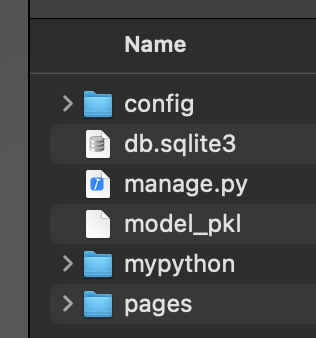
As well, you will need to **pip install sklearn in your virtual environment.**

****

Example 10: Moving Your Model to the Web

Starting with the solution for Example 8, add the binary model to the root of your web project just above the pages folder. You must also ensure that your virtual environment is activated.

Your folder contents should look something like the following:



Then update your results() function so it loads the binary model and makes a prediction with the data sent from the user input.

|  |
| --- |
| import pickle  import sklearn # You must perform a pip install.  import pandas as pd  def results(request, choice, gmat):      print("\*\*\* Inside reults()")      # load saved model      with open('../model\_pkl' , 'rb') as f:          loadedModel = pickle.load(f)      # Create a single prediction.      singleSampleDf = pd.DataFrame(columns=['gmat', 'work\_experience'])      workExperience = float(choice)      print("\*\*\* GMAT Score: " + str(gmat))      print("\*\*\* Years experience: " + str(workExperience))      singleSampleDf = singleSampleDf.append({'gmat':gmat,                                              'work\_experience':workExperience},                                          ignore\_index=True)      singlePrediction = loadedModel.predict(singleSampleDf)      print("Single prediction: " + str(singlePrediction))      return render(request, 'results.html', {'choice': workExperience, 'gmat':gmat,                  'prediction':singlePrediction}) |

Next, update your results.html file so it shows the prediction.

|  |
| --- |
| <!-- templates/home.html -->  {% extends "base.html" %}  {% block content %}  <h1>Results page</h1>  You entered the following information:<br/>  Years work exerience: {{choice}}<br/>  GMAT Score: {{gmat}}  Prediction: {{prediction}}  {% endblock content %} |

The output should show the prediction.

