PLTW COMPUTER SCIENCE

Activity 3.2.2

One-Way Communication: Blog

goals

- Incrementally develop a web app using the Django framework and *Python*
- Learn how web pages are connected and accessed

description of web App

Create an app as part of a website framework that allows a user to post blogs.

Essential Questions

1. Why are secure practices related to passwords and posting content online so important?
2. What are some arithmetic and logical concepts that are used over and over?
3. What computer science terms keep confusing me?

essential Concepts

- Django Framework
- Cascading Style Sheets
- HTML
- Databases
- Conditionals and Modulo

Resources

Custom Blog Platform

The term “blog” is shorthand for “web log”. Originally used as online diaries, blogs serve as mostly a one-way form of communication. The author of a blog, a “blogger”, may share ideas, thoughts, and feelings with the rest of the world using the internet as their medium.

There are many platforms or hosts available for blogging. In this activity you will create your own simplified, custom blog platform. Your blogging website will allow a single user to enter text into a text field and post that text to their web page. All posts will appear on a single page with the most recent post at the top.

Before you construct your blogging app, you will create a test bed app for HyperText Markup Language (HTML). Remember, HTML provides <tags> that format your pages and allow navigation on the web. In the previous activity, you mostly copied and pasted the HTML that was provided to you. Here, you will practice using some HTML syntax to format your site’s views.

Import Statements

One of the goals for this activity is for you to use import statements without being prompted to do so and to understand what those import statements provide. Import statements allow you to call functionality from other places than the file you are currently working in. Something you have done, and will continue to do, is to use the *models.py* file to define classes that will be used in your website.

The models file imports “models” from the django database (*django.db*), so you have access to all the models in that database without having to make them yourself. You can then import these models, and the models you define, in different files. To import specific classes from a models file, provide an internal location to the file using the import statement, “*from .models* (file location) *import Question* (class)”. You can import every library in the framework at once, but it will slow down your program. It is best to only import what you need from the database, and only into the files where you actually need to use them.

Getting Started

1. Navigate to Cloud9 and log in.
2. Create a new workspace using the naming convention set forth by your teacher. Example: *blogsite-lastnamefirstinitial* (Remember, Cloud9 uses only lowercase.)
3. Choose the **Django** template and create the workspace.
4. Click the **Run Project** button at the top right of the screen.
   Your framework is now running and can be shared or viewed through any web browser that has the URL.
5. Visit the URL for your new site.
6. Verify that your site presents the “It worked!” message.

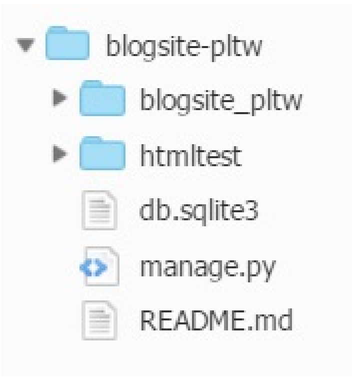
Just as you did in your *pollsite* app, you will set up a website app that uses the built-in functions of Django. You will again use temporary messages to let you know when you are viewing the index page for your site, before you make your index view more complex. You may wish to go back and reference your pollsite app to remember what this looks like, as some details have been abstracted here to focus on new app feature development.

7. Create an app in your project called *htmltest* by entering the following command in bash after the $:

   ```
   python manage.py startapp htmltest
   ```

   This code creates the basic directory structure of the polls app.

   - The top-level folder *blogsiteyourname* is the Cloud9 container or workspace.
   - The second directory level folder *blogsiteyourname* contains the *Python* package for your website.
   - The second directory level folder *htmltest* contains the *Python* files for your test-bed app.



8. Open your *htmltest/views.py* file.
9. Import the variable *HttpResponse* from *django.http*. *HttpResponse* is a class provided in the Django Framework that handles responses to view calls.

   Important: The order of the imports does not matter, as the program will import everything at the top before it starts the code below. This is why imports are put at the top of the file, so that they will be imported before they are needed anywhere in the code.

10. Define a function for an index view (request) that returns an *HttpResponse* with a test message allowing you to verify that your function is working correctly. Remember that views take a “request” as an argument.

    Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

11. Try to observe your new view running at your URL with */htmltest/* at the end. You will receive a 404 error because you have not yet registered your view in the URL configurations for your website.
12. Create a new file in the */htmltest* directory named *urls.py.* Remember, you will now see two *urls.py* files, one in the blog app, and the one that was already created for your site when you ran the *python manage.py startapp htmltest command.*
13. Add code as you did in the pollsite app to import the *url* class and create a *url* function to call the index page to the *urlpattern*. The index page is the starting point for most websites, so the code to call the index will look identical to your previous app’s *urls.py* file.

Important: Throughout this activity, you may choose to use your code from the previous polls app. Many of the files will be set up the same way, just with different file names. You also may want to have each workspace open in different browser windows or tabs to flip back and forth. Just be careful not to modify the wrong website or file.

If you do make changes to the wrong file, you may want to use the revision history option under the file menu tab to put your app back to a working version.

14. Open the *blogsite-lastnamefirstinitial/urls.py* file.
15. Add the following code to the list of URL patterns:

```
url(r'^htmltest/', include('htmltest.urls')),
```

16. Save and try to run your project.
17. Read the error message. Identify and incorporate the function you need to import so that *htmltest.urls* is included in the *url* patterns.
18. Visit your URL with /*htmltest*/ at the end and verify that you see your test message and not a 404 error.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Up to this point, these are the same steps you went through in creating the previous app. These steps are the basics for creating any app in the Django framework. Therefore, understanding them is important for any future development.

19. Complete the four parts of the activity.

Part 2: HTML TestingPart 3: Blog SetupPart 4: More ViewsPart 5: Post Titles

Part 2: HTML Testing

Playing with HTML

When a user navigates to a URL, anything in the argument to *HttpRequest* will be sent to the user's web browser for that view.

In this part of the activity, you will pass various strings of code in the arguments of the *HttpRequest* for the index view of your *htmltest* app. In essence, a user goes to a URL, and the browser requests information. The *Python* function *HttpRequest* acknowledges that the request was made and returns the argument you provided in the *()*. In this part of the app, you will pass along a string of HTML code, with other text strings, through the *HttpRequest* function, so that a browser will visually display the word "strings" based on the HTML code.

You will determine which HTML strings to pass based on examples of HTML code provided. You will make modifications to the example code to complete instructions by making what displays on your website have the same formatting as the example in this activity.

The built-in *HttpRequest* methods include GET and POST. They must be all capitals to show they are a method of the *HttpRequest* to either post information or get what is already there. A form will end with a method to post the information in the form, and afterward a user will request to get that information to view. The *HttpResponse* function has the argument in *(request)* so the function knows whether a user is posting or getting.

About HTML Tags

HTML uses a variety of **tags** to tell a browser how to **render**content. Rendering is the process of displaying visual content using computer hardware. A tag is indicated by:

```
<keyword attribute="value">Example Text</keyword>
```

The standards for HTML require that all opening tags are followed by a closing tag (or terminate with the "/" character). For example, the opening paragraph tag is <p>. To display a paragraph for your user, you need to close that tag with </p> at the end of the paragraph text.

Here is sample code for a single paragraph using <p> and </p> tags:

```
<p>  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maurisc
enim quis tincidunt placerat. Suspendisse et efficitur neque sdfjklf
. Aliquam porttitor nec libero sed tincidunt. Praesent sodales libero in e
tempor finibus. Vivamus posuere diam dui, vel fringilla
diam blandit at. </p>
```

If you open a tag but do not close it, it can cause problems for the browser, because it does not know where to stop the special formatting. It could also interfere when you try to start a different type of formatting, because the browser is still applying the <p> properties to the text.

Instead of displaying an error, the browser might ignore any content or formatting that is not closed by a tag in HTML. After you add each HTML tag, save your file, and navigate to your URL with "/htmltest/" at the end to verify that you have achieved the desired results.

Paragraph Tag <p>

PLTW DEVELOPER'S JOURNAL You will learn many HTML tags as you work through the course. As you come across each new HTML tag, record it in your PLTW Developer's Journal and describe what it does.

1. Open your *views.py* file with your temporary message.
2. Add two paragraphs of text to your *htmltest* view. Do not copy and paste from the example above. Write your own paragraph, such as things to remember for creating an app.

   Important: Remember, in *Python*, you can use a backslash character (\ ) for string continuation, so the whole paragraph does not end up on a single line in the *Python* file, but it will be output together. Be sure to enter the character in a string and press **Enter** right after the backslash symbol to let the program know that you are continuing the string.

3. Save the view and navigate to your URL with */htmltest/* at the end to verify that you have two paragraphs separated by white space.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Header Tag <h1>

Here is an example of a header tag:

```
<h1>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</h1>
```

When using a header tag, you can replace the "1" with a number (from 1 to 6) to achieve other visual effects. When you do so, make sure you modify both the opening and closing tags in the same way .

4. Add three headers, each with different numbers, to your view.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Redirect Tags <a>

Have you ever wondered what the code for a hyperlink looks like? There are two parts to a hyperlink, attributes and values. The opening < a href=" "> tag defines what the link itself should look like (attribute) and redirects a user to another URL in quotes (value). The full HTML code looks like the following:

```
<a href="https://www.w3schools.com">Visit W3Schools.com! </a>

#A browser understands the pattern from above as follows:
<open a tag HypertextReference="SpecificURL">Text string to display!</clos
```

Attributes modify HTML tags by providing either needed functionality or other modifications. Attributes must be paired with values. Each attribute has different kinds of values.

In the link tag above, the <a> HTML tag includes an attribute and the *href* attribute has the value "https://www.w3schools.com". The link tag cannot function without the *href* attribute. The tag above will display "Visit W3Schools.com!" to the user in hyperlink styling, typically blue text, unless you specify a different color using an HTML attribute or CSS. When the user clicks the link text, it will redirect the user's browser to the URL specified in the attributes of that tag.

5. Add a link tag to your view that displays the text "a neat website" and redirects the user to a different web page. Some things to consider:
    - The attributes use double quotes (" "), so it might be best to use single quotes (' ') for your *Python* code.
    - The *Python* code reads in sequence; once you start a string with " , *Python* takes the next instance of " as closing that string, even if you intended it to define an attribute.
    - When programming in *Python*, using single quotes for strings and double quotes for attributes helps to reduce code errors.

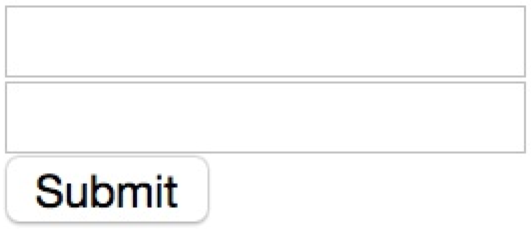
Form Tags <form>

With the *form* tag, you can create a web form and add some of the form elements with which you are familiar, such as text boxes and buttons. The form you create in this part of the activity will have *Python* code added later to store data from forms. Before you can store the information, you need to know how to create those user interfaces in a database and what you want them to look like. Here is an example of code for a form with a text box.

```
<form><input type="textarea" /></form>
```

There are many options in HTML to create custom forms and fields. Many programmers do not have them all memorized; they just know how to find what they need quickly.

6. To practice this skill, use the internet to find HTML that will create a form that looks like the following with text areas and a button in a single form:



Hints

- You might find the tags you want at: https://www.w3schools.com
- You may use any tags you like.
- You only need one form element with all the form fields inside and just one opening tag and one closing tag.

Review HTML

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

PLTW DEVELOPER'S JOURNAL Take some notes on HTML and how it works with *Python* code. Answer the following questions based on the code:

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse('p>My first paragraph is just sentence./p>')
```

1. What is *index()* ?
2. What is *request* ?
3. What two methods might be seen in the argument?
4. What is *HttpResponse()* ?
5. What is the data type of what is returned in *HttpResponse*?
6. What is <p> called?
7. For HTML to work in *Python*, what needs to be outside of the HTML tags?
8. Adding the ' ' symbols outside the tags turns everything into what?
9. Explain when to use (" ") versus (' ') .
10. Where does the *HttpResponse()* functionality come from?

7. Continue to the next part of the activity.

   Part 2: Blog Setup

Part 3: Blog Setup

Overview of Blog App

You created the *htmltest* app to learn about HTML tags and explore what some of them look like on a web page. You will now create a second app in your project (blog). You may use this blogging app later in your final project and problem.

Rather than using step-by-step instructions to set up your new blog app, create it using the same process you did for the *htmltest* app and your previous *pollsite* app. If you need to, you can always refer to earlier steps in this activity, or look back to the *pollsite* app instructions to help you figure out what to do. The goal here is for you to try the steps on your own, so start by thinking through what you need to do before you do it. Use the next section to think through creating your new blog app. Once you are feeling comfortable with your plan, create the blog app.

Remember that each view of a web page in a Django application has a specific purpose. The Index view, used in both applications you have created so far, is the starting view and home page for both applications.

PLTW DEVELOPER'S JOURNAL Identify at least two views that will be needed for your blog app. You may brainstorm and come up with more.

Set Up the Blog App

1. In bash, create a new basic directory for the blog app. This time you will name it "blog".
2. In your *blog/views.py* file, import code from the framework to handle *HttpResponse*s.
   - In your *blog/views.py* file, define a function for an index view.
   - Create a temporary HTTP response message to let a reader know they are viewing the index page for your blog.
3. Create a new file in the */blog* directory named *urls.py .*
4. In the *blog/urls.py* file, register the URL (map the function call to your index page).
   - Do you need to add "include" to the *url* configuration import?
   - Do you need to add a function call for the blog to the URL patterns of the site?
5. Save and visit the index page in your browser to confirm the temporary message is visible.

Creating, Activating, and Migrating Models

PLTW DEVELOPER'S JOURNAL Use the assessment below to help check your knowledge about the order of setting up a model and record the best notes you can.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

In your blog, you will allow a user to post a message. To do this, you need to create a model for that post. A *Post* has two fields: the text of the post and a publication date.

6. Add the following code to *blog/models.py*:

```
class Post(models.Model):
    post_text = models.CharField(max_length=1000)
    pub_date = models.DateTimeField('date published')
```

7. Look at the *models.py* file from your previous *pollsite* project. How does the Post class in *blog*/*models.py* vary from your Question and Choice classes in *polls*/*models.py*?

   Important: Identifying the similarities and differences will help as you set up your own models in future apps.

8. So that you can access your models, tell the website server that the blog app is installed.

   - Navigate to the *blogsite-lastnamefirstinitial*/*settings.py* file. (Remember, your website and filename are unique.)
   - Add *'blog.apps.BlogConfig'*, as the first line in the list of *INSTALLED_APPS.*

   ```
   'blog.apps.BlogConfig',
   ```

   Important: Remember to add a comma at the end of line, check the indent, and save the file.

9. To let Django know that you have added a new model to your code, make a migration. Enter the following command in bash:

```
python manage.py makemigrations blog
```

   Important: The migration will look at the fields in the *models.py* file. The migration algorithm will compare what is already set up in the database with what is in the *models.py* file. Then, the algorithm will add or delete fields (like title, text, and date), so that only what is outlined in the *models.py* file is in the database.

10. To set up your Question/Response database, enter the following command in bash:

```
python manage.py sqlmigrate blog 0001
```

Django will set up the tables for your database with "blog_post". It names the table "blog_post" because the app name is "blog" and you have just added a "post" class. Table names only use lower case letters.

11. Navigate to *blog/admin.py* and add the following code and save the file:

```
from django.contrib import admin
from .models import Post

# Register your models here.
admin.site.register(Post)
```

12. Setup the Admin Web Portal to test your blog.

    - *From django.contrib import admin*
    - *From .models import Post*
    - Register your models with: *admin.site.register(Post)*
    - Using the credentials you created in your previous app, log in to the Django administration portal. If you did not write down your username and password, you can create a new superuser account with:

    ```
    python manage.py createsuperuser
    ```

    Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

13. Run your project and view the admin page.

    - To view the admin page, run the project and click **Your code is running at https….**
    - Add “/admin/” to the end. If your web browser is still open, just replace “/blog/” with “/admin/”

14. Log in using the credentials you just created.

    Important: If blog fails to appear in the administration portal, what did you most likely forget to do in the last few steps?

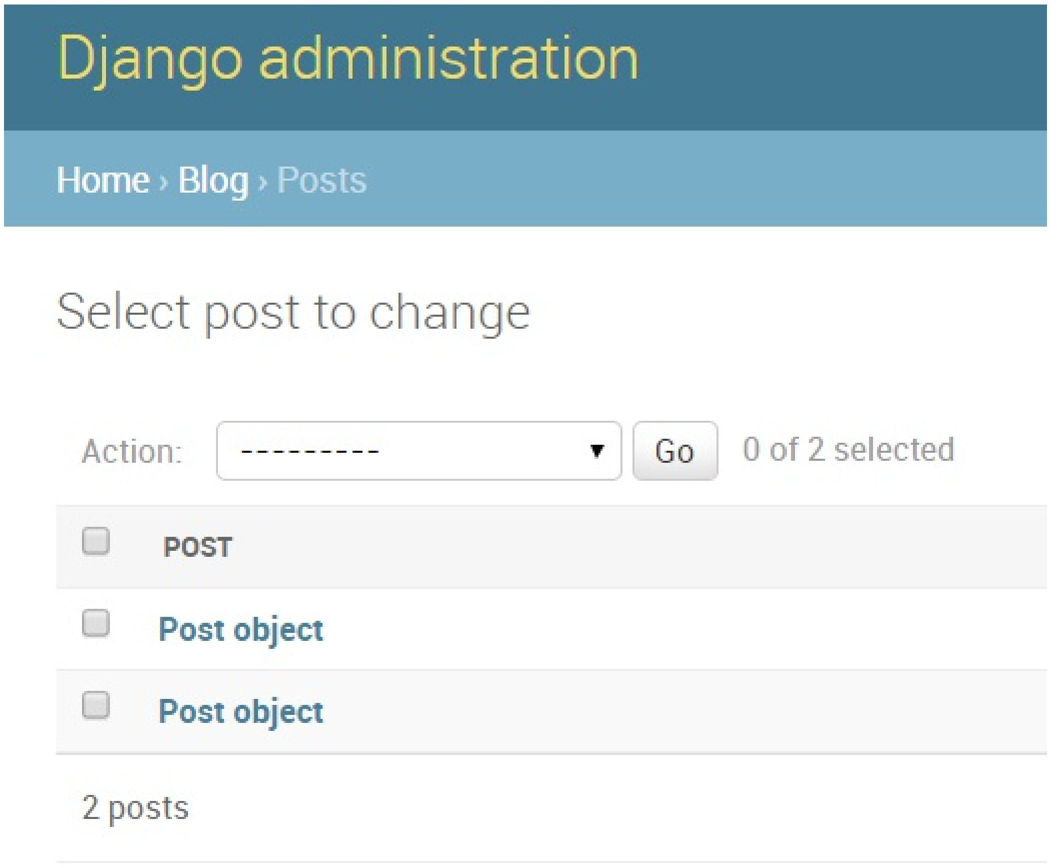
15. Click the **+ Add** for the Posts data under the BLOG app. You will be prompted to:

    - Add text to the “Post text” field.
    - Select a “Date published”.
    - Select **Today** and **Now**, otherwise you will not be able to view the post until the date and

time you select in the future.
- Save your post.

  Important: If you encounter errors that will not let you post, you may need to stop your application, make sure everything is saved, check that you did both migrations, and then restart the server.

16. Verify that you just created a Post object in the admin portal.
17. Create a second *post*. Remember that there are no identifiers in the admin framework for which object is which.



Making Web Pages Dynamic

Currently, you have only one index view for your website. The view displays the same message you coded in the *HttpResponse* function, regardless of what content is added to your database. In this part of the activity, you will allow your user to view information from your database, making your website behave dynamically. Dynamic websites respond to user interactions differently depending on the data in their databases.

First, you will modify the index page for your blog so that it shows a list of all of the *Post* objects in your database. To test this functionality, you need to have multiple Posts in your database.

Remember, the *index(request)* function is an argument, and an *HttpResponse* is what gets returned. What file contains a list of all the pages that can be called and the *HttpResponse*s you want to see on those pages?

Right now, our index page has a placeholder message letting us know we correctly navigated to the page in our web browser.

18. Navigate to the file with the placeholder message. Remove the temporary message and create or modify code, so that each blog post will be displayed in order from newest to oldest.
19. Modify the following code, so that your blog will display each of the posts in the database. Use the reflection questions and previous sample code to help you set up your index view.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

PLTW DEVELOPER'S JOURNAL Think through the following questions to help you set up the index view:

1. In the previous app, the classes or models you created were *Question* and *Choice*. What is the name of the class you created in this app ?
2. Is *BlogApp* the same as *blogapp* in *Python*?
3. In the *views.py* file, how did the code know anything about the model *Question* in a different file?
4. What will you change the variable *latest_question_list* to?
5. What will you change the *Question.objects.order_by* to*?*

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

If you view your blog posts right now, you will see all the posts running together. This is the same problem you experienced in the *poll* app before you used HTML to change the browser rendering.

20. In the text field of each *Post* object in the Django admin tool, add bold <b> tags and language that let you see when the first post starts.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

You need to format the code in the *views.py* file and not in the actual blog post. Rather than use the output line as written previously, you could create a local variable (*output*) as an open string, and increment the post list just as you did in the previous app. Using this method, it is possible to add additional line items to the output, such as a loop:

```
Output = ''
```

```
output + .join([p.post_text for p in post_list])
```

21. Delete the tags in the Post objects.
22. Add the variable and *for* loop to your index view function.

    ```
    output = output + .join([p.post_text for p in post_list])
    ```

23. View the index page of your site. It still looks a little unprofessional and unreadable with posts running together.

24. Create or modify the code in your *blog/views.py* index view function to:

    - Create a variable *output* as an open string.
    - Increment with each new post.
    - Add paragraph tags and concatenation to separate each new post.

    Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

25. Save your code and visit your site's URL with */blog/* at the end.
26. View the page's source to verify that your opening <p> and closing </p> tags appear where you want them.

    Note: In Chrome you can see tags by selecting **View > Developer > View Source** or by right-clicking in the window and selecting **View Page Source**.

27. Test and verify that:
    - ☐ Your website is still running without errors.
    - ☐ The website can access the Post models and display posts on your index page.
    - ☐ New Posts have the same formatting as the other posts.

28. Continue to the next part of the activity.

    Part 3: More Views

Part 4: More Views

Attributes and Views

In the last activity, you allowed users to update existing Choice objects in your database through your website. Now you will allow the users to create Post objects in your database through your website.

1. Navigate to the *blog/views.py* file (if you are not already there).
2. Create a view function named submission.
3. Have your submission function return a temporary *HttpResponse* string letting you know that you are viewing the blog submission page.
4. Register this new *urlpattern* (submission view) in your URL configurations.

   Important: Remember, there are two URL configurations files, one for the entire site and one only for the app.

   Which would you want to add the submission *urlpattern* to?

   Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

5. Verify that your submission page is active. Keep in mind that in the directory where you check, you will need to add "*/blog/submission/*" to the end.

PLTW DEVELOPER'S JOURNAL What is happening in the *urlpatterns* that allows you to see the views at a specific URL?

Tag Attributes and the Submission View

Now you are going to combine all the things you have learned, and vocabulary is a key part of that. As you move through the next part, be sure to update your TEMP charts to give you a reference as to how the new code constructs would look in *Python* and HTML. This section explains how you will create and format a submission form for people to post and get the posts.

HTML tags may have attributes, which in turn have values. Attributes are somewhat like variables. They communicate values that a web browser can use to determine how your page should behave. Values are typically enclosed in double quotes.

You saw an attribute and value pair such as *href*="/*polls*/" in the last activity. In the following example identify the parts, such as class, tag, attribute, value:

```
<a class ="button" href = "/polls/submission/"> Submit a post </a>
```

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Next, you will create the *form* view. To direct the browser, you need to use the form tags. A form must begin and end with a form tag. The opening form tag needs to assign values to two attributes: action and method. In the code below, action and method are the attributes; "/polls/" and "get" are the values of those attributes.

```
'<form action = "/polls/" method = "get">'
```

When a form is submitted, it needs a destination to redirect the web browser to perform the paired action. The two most common HTTP methods are GET and POST. Whereas GET is the default method used to simply provide content from a server to a client, POST allows the client to modify data on the server in some way.

In this case, you will modify data on the server by creating a new Post entry. Unlike the example above, you will direct the program to your URL with /*blog*/ at the end, which is your index view. When Django performs this redirection, it needs to do so using the POST HTTP method. This will take all the information in the form and post it to your *submission* view.

6. Review more information on GET and POST.

Each element in a form is described by a single input tag with attributes like *type*, *name*, *id*, or *value*. A good strategy for building the *submission* view is to create an output variable and keep adding to it (incrementing and concatenating). This way, you provide a single variable (output) to the user through the return function that has all the information you want the user to see.

7. In the *views.py* file, modify your submission function.
   - Create a local variable *output* in your index function as you have done previously.
   - Use the code below to help. Fill in the blanks to help you construct the opening form tag for the *submission* view to post information to the *index* view:

   ```
   Output = ''
   output = '<form _________ = "/blog/" __________ = "post ">
   ```

   The submission form will add to the blog. A user will click the **Submit** button and be redirected to the /*blog*/*submission*/*value* of the URL (action attribute). The program will then submit this data using a built-in POST method to add the user’s response to your server database.

8. Create a text box form element using the <*input*> tag. This text box is where the user types their blog post.
   - You want the type of input for the box to be *textarea*.
   - Name the form element “posttextbox”.
   - Name the id “posttextbox”.

- To terminate an input without a closing tag, include a forward slash before the right angle bracket of the opening input type tag.
- The *form* element will have three attribute pairs:

```
<input>

type = "textarea"
name = "posttextbox"
id = "posttextbox"

output = output + '<input type = "textarea" name = "posttextbox" i
```

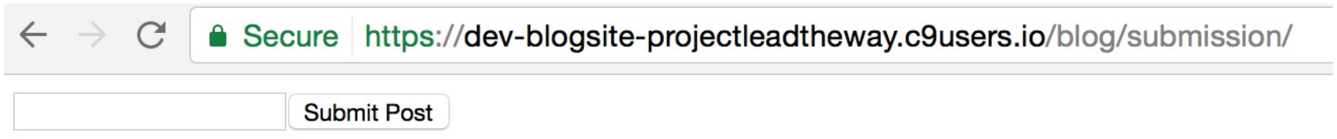
Important: In the code above, note the / character with a border around it. The / at the end will close the input tag in the same place it is opened to avoid having to add a separate closing tag.

9. Create a submit button form element.
   - Create an output variable.
   - Add to the output variable another form element with a submit type and the value Submit Post.

```
input type = “submit”
value= “Submit Post”

output = output + '<input type = "submit" value = "Submit Post" />
```

10. Now concatenate a close form tag *</form>* to the output variable. Remember, you opened the code with a form tag, so you need to close the tag.
11. Visit your URL with */blog/submission* at the end. You should see this:


Secure https://dev-blogsite-projectleadtheway.c9users.io/blog/submission/

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Cross-site Request Forgery

Cross-site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions in a web application in which they are currently authenticated. The CSRF error will come up with pages, so you need to have an exemption added to the code to avoid the error.

12. Explore more about Cross-site Request Forgery (CSRF).
13. In *views.py*, add the following code in the appropriate places to bypass CSRF considerations as you did in the last app.

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
```

Index View and Multiple Posts

You will now alter your index view function to handle the POST that it receives from the submission page, to save the title, message, and date of the POST.

You will use an *if else* conditional to determine if the method field of request in your index view is "POST". If the request is POST, the conditional should save the form information to the database. If it is not POST, it is GET, and the conditional should display the information from the database.

14. Import *datetime* to your *view.py* file. This is a *Python* object, not a Django library, so all you need to enter is “import datetime”.
15. Create the code for the *if* statement. Your *else* statement should be the original code already written.
    - Add the conditional to evaluate whether the request method was "POST":

      ```
      if request.method == "POST":
      ```

    - Construct a new *Post* object and store it in a variable. Name the variable *user_submission*, as shown:

      ```
      user_submission = Post()
      ```

    - To set the *post_text* field of your new variable to the value contained in the request’s POST field, use the ***__getitem__*** function with argument *"posttextbox"*.

      ```
      user_submission.post_text = request.POST.__getitem__("posttextbox'
      ```

    - To set the *pub_date* field of your new variable to the value retrieved, call the *now* function with no arguments on the *datetime* field of *datetime*.

      ```
      user_submission.pub_date = datetime.datetime.now()
      ```

    - Call the *save* function with no arguments on your new variable.

      ```
      user_submission.save()
      ```

16. Set your output to a string that will thank the user for their submission.
17. Visit your URL with */blog/submission/* at the end.
18. Enter some text in the text field and click **Submit Post** to verify that you are redirected to your URL with */blog/* at the end and that it displays a thank you message.
19. Refresh your browser.

You should see your Post objects, with the most recent ones at the top. If you do not, check your code against the following sample solution code, correct any issues, and make notes of differences in your PLTW Developer's Journal. Be sure to read the comments to understand what each line of code is doing.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Additional Features

To increase the usability of your blog app, you will now add some features. A blogger would not want to type a URL to navigate between viewing posts and making them. Similarly, people who read a blog typically want to see a title and a timestamp for the posts to give them some context. Using what you have learned about HTML, you will add these features by modifying code from your pollsite app.

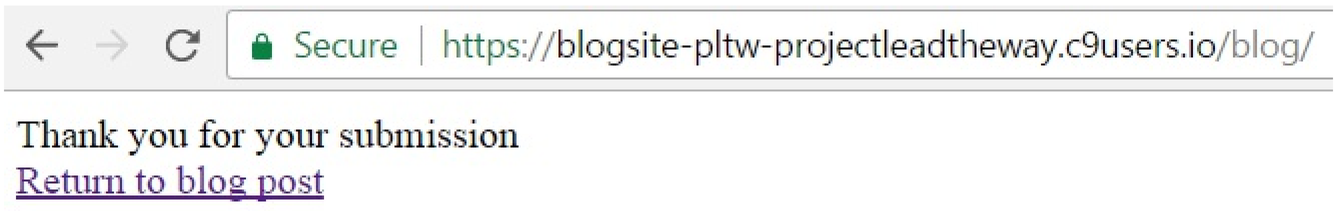
Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

20. Add a link at the top of your index view that takes the user to the submission view. This is the page where users view blog posts. If the page does not look the way you want, you might need to add a line break tag <br /> or other tags you have learned about.

    Important: To navigate to the submission page, a user selects the **Submit a blog post** hypertext. Review the difference between hypertext and hyperlink.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.	Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

21. Using a similar anchor tag, add a link at the top of the submission view that takes the user back to the index view. (This is the page the user sees after they submit a post.) To navigate back to the index page, the user should be able to click the **Return to blog post** hypertext.



Thank you for your submission
Return to blog post

22. Verify that the links allow a user to navigate between the website pages.
23. Continue to the final part of the activity.

Part 4: Post Titles

Part 5: Post Titles

Migrations Revisited

Your Post data type originally contained only two pieces of information—the text of the post and the date it was published. To complete this app, you need to add a title field to the Post class. Just because you add the title field to your Post class in *Python* doesn't mean your database will have storage space specified for it!

Django has a migration tool to update your database when you make changes to your model code. You already used migrations to set up your initial database. Creating and running a migration will create a new database. The structure will automatically accommodate your new model and copy all your data into it.

1. In *models.py*, add a new field to Post called *post_title*. Make it a *CharField* model with two arguments separated by a comma.
    - The first is a *max_length* of 70.
    - The second argument should be default, with a value "Old Title".

   Important: If you get stuck, review the earlier parts of the activity terms and concepts for argument and value.

2. In bash, run *python manage.py makemigrations blog .*

   ```
   python manage.py makemigrations blog
   ```

3. Confirm that the migration was created with the output in bash. If you made a mistake, you will see the following message:

   You are trying to add a non-nullable field 'sample_field' to post without a default; we can't do that (the database needs something to populate existing rows).

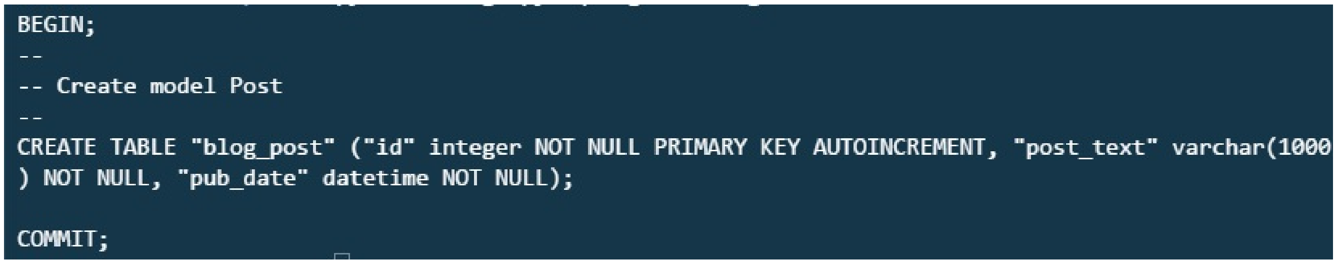
   Select a fix:

    - 1. Provide a one-off default now (will be set on all existing rows)
    - 2. Quit, and let me add a default in *models.py*

   Type the number "2" and press **Enter**. Then verify that your *post_title* field is correct in *models.py*, save your file, and try this step again.

4. In your blog folder, open the migrations folder to see the file *0001_initial.py*, which is the first migration you created at the beginning of the activity. Depending on how many tries it took to create your latest migration, you may have several files in this folder. The latest file will always have the highest number in its prefix.

5. In the bash command line, type “python manage.py sqlmigrate blog” followed by the number of your most recent migration and press **Enter**. The message that’s returned shows a series of MySQL commands that end with a COMMIT;



```
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "post_text" varchar(1000
) NOT NULL, "pub_date" datetime NOT NULL);

COMMIT;
```

Important: All your old posts now have a *post_title* attribute set to "Old Title".

Incorporating the Post Title in Your Views

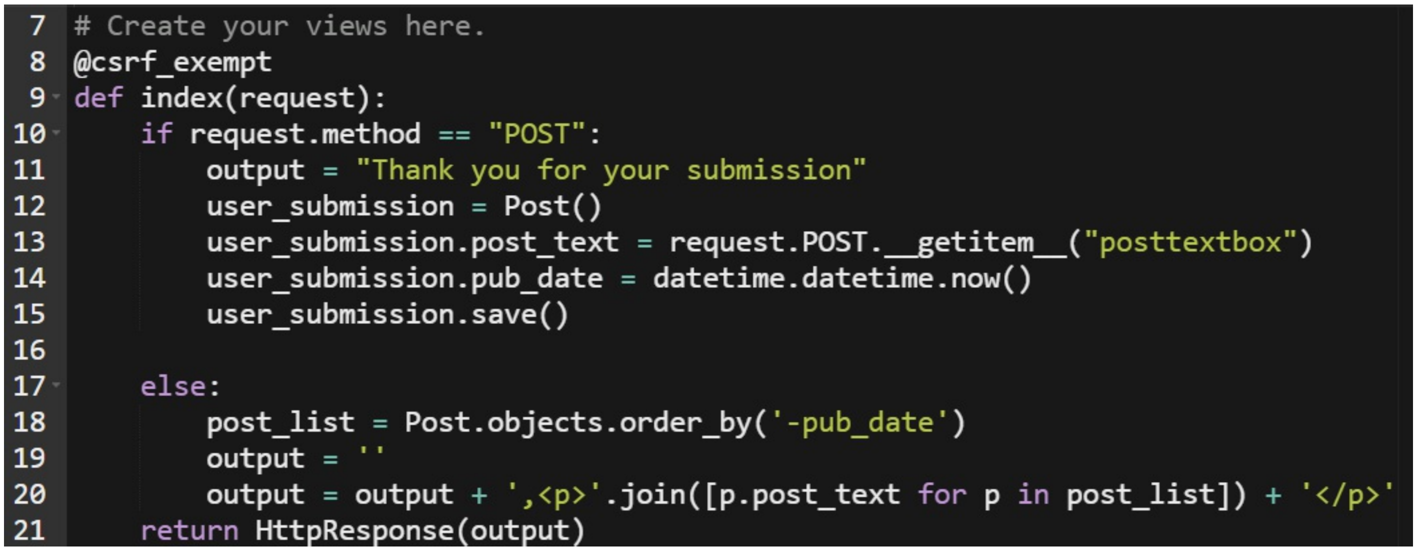
To make post titles part of your site, you need to add them to your index and submission views. Each post on the index page should include the title and the publication date, and the submission form will need an additional text field where the user can type their post title.

Making Your “for loop” More Obvious

In database tables, the information that is stored for each entry all has the same data categories. In this blog app, the categories are post title, text, and date. To get and display those entries from the database, you create a loop that moves in a sequence to pull out all these categories of data for one entry, then the next entry and the next, until you have all the entries you want.

There is already a loop in this program; however, it looks different from other loops you have seen. In line 20 of the code below (your line numbers may be different; always look for the function, not the line numbers in your own code), you are using a *for* loop to pull the post text for each item in a list. This is fine for pulling one item from the list, but now you are going to pull three items:

- Post title - *p.post.title*
- Publication date - *p.pub_date*
- Text of the post - *p.post_text*



```
 7 # Create your views here.
 8 @csrf_exempt
 9 def index(request):
10     if request.method == "POST":
11         output = "Thank you for your submission"
12         user_submission = Post()
13         user_submission.post_text = request.POST.__getitem__("posttextbox")
14         user_submission.pub_date = datetime.datetime.now()
15         user_submission.save()
16 
17     else:
18         post_list = Post.objects.order_by('-pub_date')
19         output = ''
20         output = output + ',<p>'.join([p.post_text for p in post_list]) + '</p>'
21     return HttpResponse(output)
```

The way *Python* describes this *for* loop in a list might look strange, but if we rearrange it, what is happening might be more obvious.

6. Delete your loop line, as shown in line 20 of the image above and replace it with:

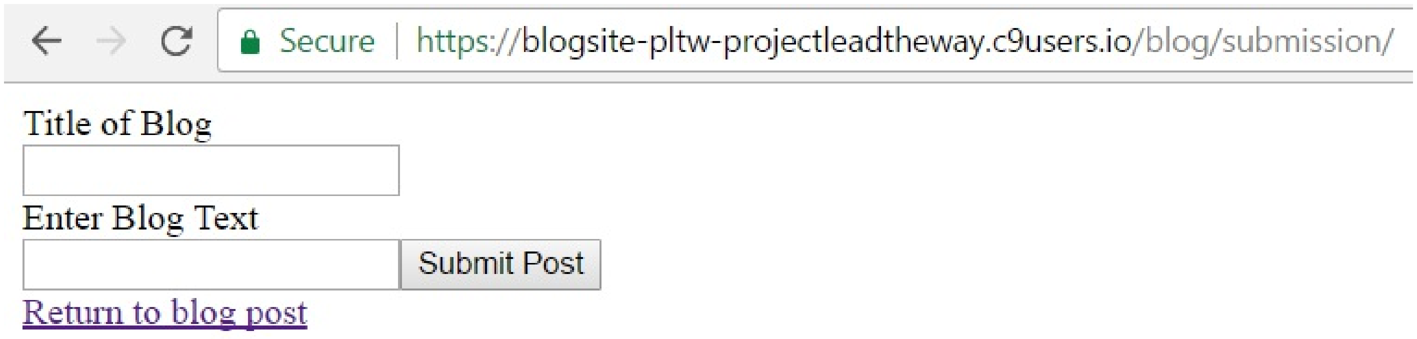
   ```
   for p in post_list:
   ```

7. Inside the loop, use HTML tags and *Python* code to display the title, text, and date for each post in the *post_list*.
8. After each step, verify that your code does what you expect it to, and when appropriate, addresses error messages.
9. In the *for* loop of index view, for when the *request.method* is not "POST":
   - Add a call to get the *post_title* in the loop so the title will display with the text.
   - Add the publication date to your index view following the post title for when the *request.method* is not "POST".
10. When you save and test your code, you will get an error. Look at the error message.
    - Why did the code have a problem with one of the fields?
    - How can you fix it?

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Creating Titles

Right now, every post has the title "Old Title". You will create an input for the user to create and submit a title for the blog. Look back at how you did this for the blog text submission in your other website. Create a text field and Submit button for the title of the blog.


← → C 🔒 Secure | https://blogsite-pltw-projectleadtheway.c9users.io/blog/submission/

Title of Blog

Enter Blog Text

Submit Post

Return to blog post

11. Add a *textarea* with name "posttitle" and *id* "posttitle" before the "posttextbox" in your submission view.
12. Add some text to help the user identify which *textarea* is for which piece of data.
13. Use HTML tags to help format the index view to be readable.
14. For when the *request.method* of the index view is "POST", store the post title in your *Post type* variable as you did the *post_text* .

Remember that any previous posts will have the title “Old Title”, but new posts should have the title the user entered in the submission view.

15. Move on after you can see the following in the index view for new posts:
    - ☐ Title as entered
    - ☐ Publication date
    - ☐ Post in a format that is readable

CSS

In this part of the activity, you will set up the style of your website using a cascading style sheet (CSS) that will apply to the blog app, as you did for the pollsite app in a previous activity. This time, you will experiment with making modifications to the CSS file to customize the view.

16. In your blog directory, create a new directory folder named “static”. The static folder notifies Django that these files will be used in the site, such as images and CSS files.
17. Within the static directory folder, create a new folder named “blog”.

    Although the naming may seem repetitive, if you had multiple apps in your site, this directory would be used to distinguish which apps the CSS is applied to.

18. Upload and drag the *style.css* file into the blog/static/blog folder in the directory structure.

    The CSS file has been set up for you, so you can look at the formatting and modify it to fit your needs later in this activity.

19. Add an import call for *static* in the *blog/views.py* file so the files can access the CSS file you uploaded:

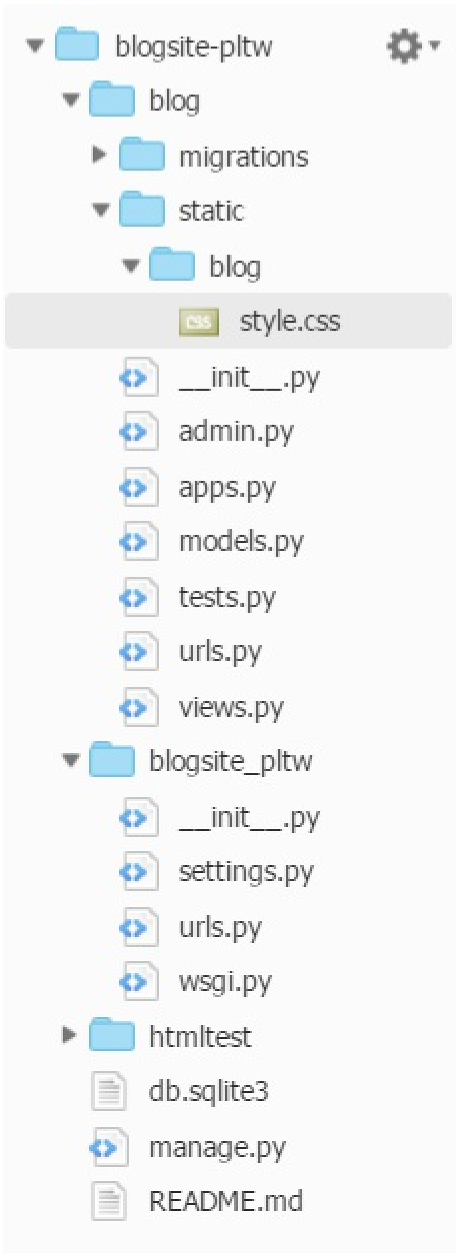
```
from django.contrib.staticfiles.templatetags.s
```

20. In each of your view functions in *blog/views.py*, prepend (add data to the front of another piece of data) and concatenate the following code to the front of your output string before its return as an *HttpResponse*:

    ```
    '<head><link rel="stylesheet" href="' + stati
    ```

    Important: You may look back at your pollsite app CSS to recall how to do this.

21. Test your site to make sure the stylistic elements have been applied to the content. Your pages should look similar to the following:

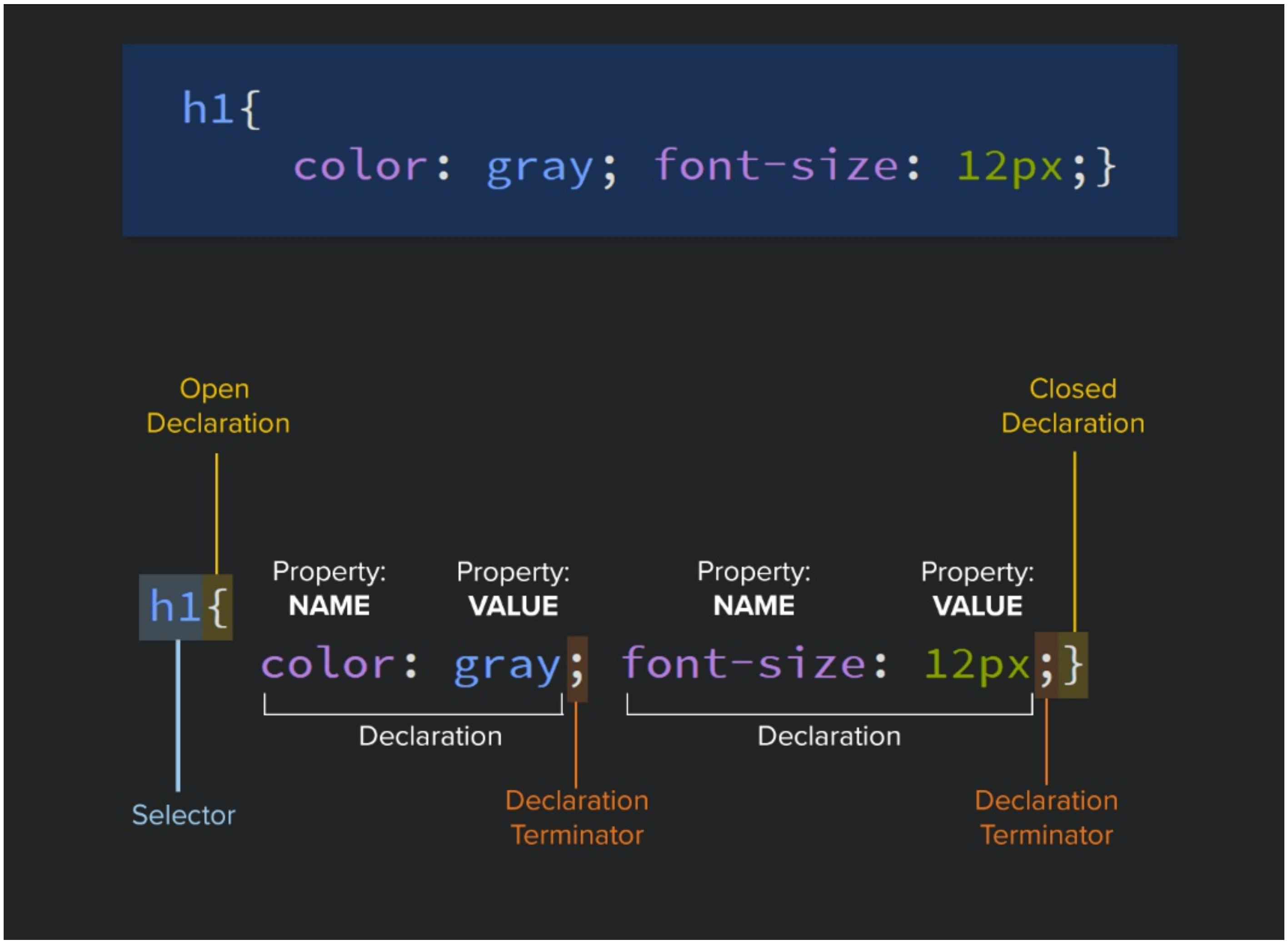


Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

CSS and HTML for Styling

At this point, your code and similarly the screenshots shown above, do not reflect all the styling contained in the provided CSS file. You will modify your HTML to take advantage of some additional styling now. In the process, you will begin to learn about the syntax and capabilities of CSS.

In CSS, statements are called “rule sets” and are made up of a selector followed by a block of declarations. A selector is an HTML element, and each declaration within the curly braces provides a property-value pair terminated by the “ ; ” character. Each property-value pair ends is connected with a “ : ” character.


```
h1{
    color: gray; font-size: 12px;}
```

PLTW DEVELOPER'S JOURNAL Open style.css and look at lines around 143, starting with /* PLTW branding */ to complete the assessment below. Record information about CSS styling.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

A title gives users a quick understanding of what they are looking at. To add a title that stands out to a user, follow these steps:

22. In your index view, add the text "My Django Blog Site" so that it is the first text that appears in your index view. Do not put it inside the header tags in the output string.

PLTW DEVELOPER'S JOURNAL Add a note describing how the text appears on your website, including positioning and any stylistic notes.

23. Modify the code you just added so that it is enclosed in a tag. How does this change the look of the words from the previous step?
24. Refresh the index view of your site.

PLTW DEVELOPER'S JOURNAL Was the effect of the header selector on your text what you expected? Why or why not?

In CSS, when multiple selectors appear in a rule set, that rule set is only applied to HTML elements that match ALL of those selectors. It is important to plan which tags overlap and enclose other tags to create the appearance you want. To test the order of enclosing tags, you are going to add an image to the header with the text you already have.

25. Within the header tag, enclose your "My Django Blog Site" in a span tag.

    Record your prediction for how this will change your message's appearance:

26. Upload the *PLTW_logo.png* from the activity's source files.
27. Place *PLTW_logo.png* within the blog/static/blog directory.
28. Add the following HTML to your index view between the opening header tag and the opening span tag:

```
<img src="../static/blog/PLTW_logo.png">
```

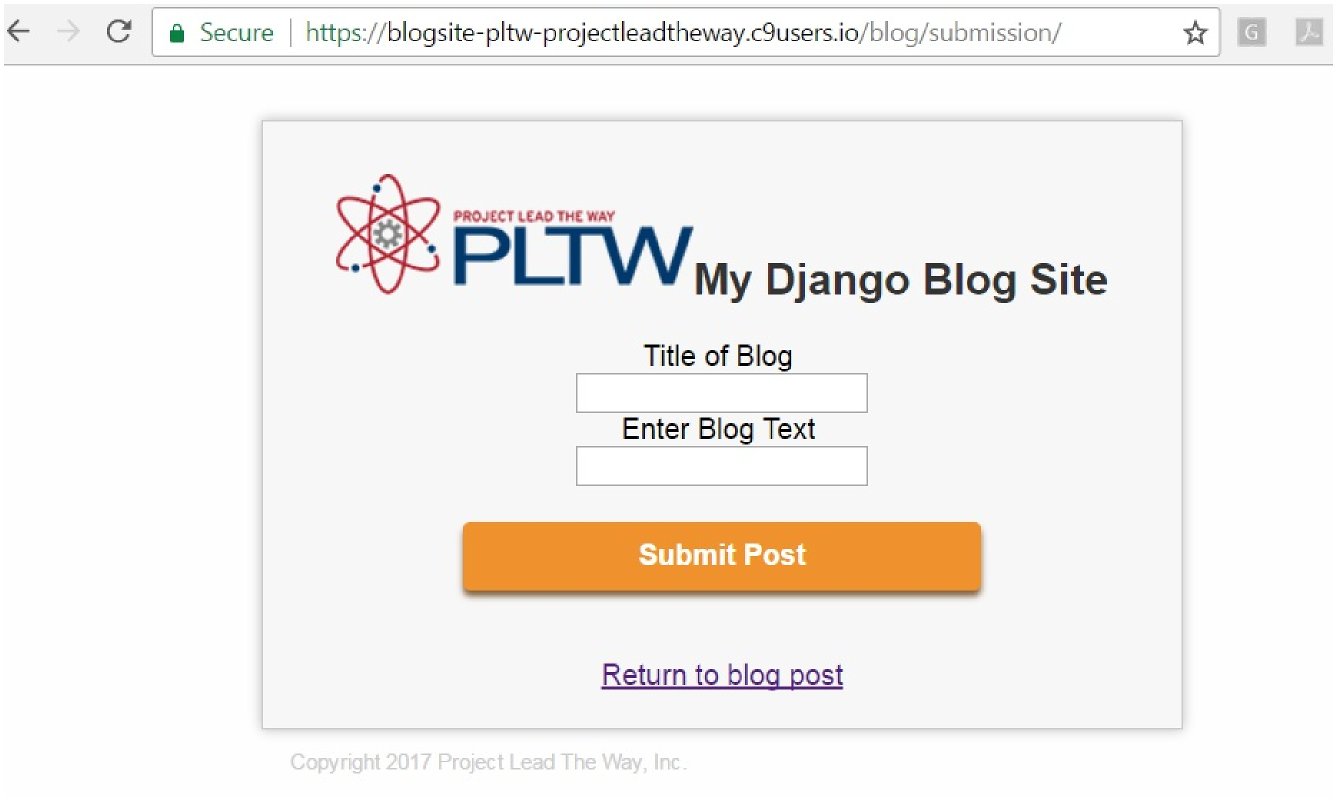
29. Navigate to your index page and see whether your predictions were correct. If not, look at the code in the CSS file and the HTML in the *Python* code to determine why it looks the way it does.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

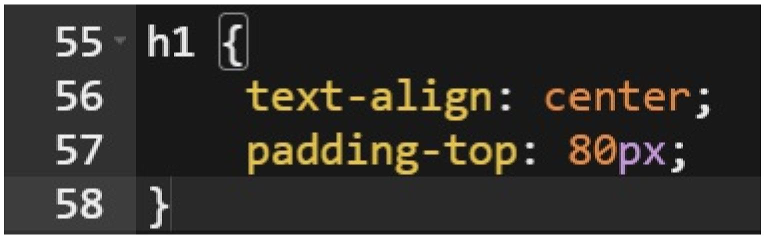
30. Modify your submission view to include the following header:

```
<header><img src="../../static/blog/PLTW_logo.png"><span>My Django Blc
```

31. Test your site to make sure the submission view has the same header as the index view. Yours may look different based on what HTML tags you applied, but a sample version looks like:


Secure
https://blogsite-pltw-projectleadtheway.c9users.io/blog/submission/
PROJECT LEAD THE WAY
PLTW
My Django Blog Site
Title of Blog
Enter Blog Text
Submit Post
Return to blog post
Copyright 2017 Project Lead The Way, Inc.

32. In the *style.css* file, find the h1 selector.



```
h1 {
    text-align: center;
    padding-top: 80px;
}
```

33. Predict what you think this selector will do to text enclosed in an h1 tag.

34. In your index view, where the *request.method* is "POST", enclose the "Thank you for your submission" message in an h1 tag and save the file.
35. Refresh your page, submit a post, and see whether the rule you set for h1 did what you expected.

The Class Attribute, HTML, and CSS

You can apply the class attribute to any HTML element! This comes in handy to quickly style your web page. For example, you may want to have different types of paragraphs look different, such as an information paragraphs versus specific steps to follow.

The following are examples of how you might write HTML to change appearance:

```
<p class="firstparagraph">this text could be styled by a CSS ruleset </p>
<p class="differentkindofparagraph">this text could be styled by a CSS rul
<input type="submit" class="akindofbutton" value="Styled Button!" />
```

With class attributes, you can specify styles for specific tags within your pages. For example, if you want some paragraphs to appear in blue text and others in red, the following CSS would accomplish this:

```
.firstparagraph {
    color : red;
}

.differentkindofparagraph {
    color : blue;
}
```

36. In the CSS file, look at the button declarations.

    PLTW DEVELOPER'S JOURNAL Choose three properties to research online and record what they should do to elements that have the class attribute "button".

37. Add the *class* attribute to each of the links in your website. Give that attribute the value "button".
38. Describe what the button class did to your links. What if you also want it to be centered?
39. In the CSS file, find a rule set that will center the links.
40. Enclose the off-center links in the tag(s) that correspond to the rule set that centers text.
41. In the submission view, you have an input element with type "textarea".

    Look at the CSS and describe how you think this input element would change if you used a textarea tag instead of an input tag.

42. Modify your input element of type "textarea" to use the textarea tag instead of the input tag. Remove the type attribute-value pair and add a closing textarea tag.

    ```
    <textarea attribute = "ValuePair"> What a user sees <CloseTag>
    ```

43. Check to make sure the submission of a post still works after you modified the HTML.

    PLTW DEVELOPER'S JOURNAL Look at the text in the submission page URL; where does it look like that text style comes from, the HTML or the CSS?

44. After you have the title input working, modify the text input, too.

PLTW DEVELOPER'S JOURNAL Update your notes about CSS and HTML tags.

Getting Fancy with the CSS

45. For some really fancy CSS styling in the not-Post index view, enclose the post title, publication date, and post text in a single article tag.
46. Enclose your post title and publication date in an h2 tag.
47. Find a rule set in the CSS file that will make the publication date for your posts align to the right side of the article box and appear in a less visually striking font. Before putting it into your code, you may look at the code and modify it here first:

48. To apply that rule set to your publication date, add a span tag with that class attribute-value enclosing your publication date.
49. Test your ability to modify a CSS file by changing the color of your links to green.

    Important: You may need to clear your browsing history and refresh your site to see this take effect.

50. Make your site look as much like the following views of a finished version of the blog site as possible.

    Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

51. After you have completed your site, discuss with you teacher what you might modify and personalize using the CSS and HTML.

Conclusion

1. How did you use functions in this activity?
2. How did you use conditionals in this activity?
3. How did you use lists in this activity?
4. How did you use loops in this activity?
5. How did you interpret and respond to the essential questions? Capture your thoughts for future conversations.