SETTING UP A PROJECT:

Creating the virtual environment:

A virtual environment (venv) is a place on your system where you can install packages and isolate them from all other Python packages.
Separating one project's libraries from other projects is beneficial and will be necessary when we deploy LL to a server.

We set up a new directory (learning_log) for our project and use the venv module to setup a virtual env:
In command promt, in the right directory, we enter the following to create a venv called ll_venv:

```
python -m venv ll_venv
```

We activate the env:

```
ll_env\Scripts\activate
```

Deactivate using 'deactivate' in cmd.

Installing Django:

pip install django
Keep in mind that Django will be available only when the ll_env is active.
Name of venv in parentheses in terminal shows that your venv is active.

Creating a Project in Django:

Tell Django to setup a new project in console:

```
django-admin startproject learning_log .
```

Don't forget the dot at the end,
it creates our project in a beneficial structure.

Django has created a new folder (directory) called learning_log containing some py-files and manage.py.

Creating the Database:

Django stores most information in a database, we create one in console using the manage.py:

```
python manage.py migrate
```

Modifying a database is called migrating.
When entering migrate command first time, it creates an SQLite database for us.

Viewing the Project:

Using...

python manage.py runserver

Django starts a development server, so you can view your project on your system to see how it works.

You can see if there were any issues and Django will give you the localhost URL as well.

STARTING AN APP:

Django projects are individual apps that are working together to create a whole.
For now, we'll use one app to do most of the work.
When the server is running and we are in the venv, we create a new app in console:

python manage.py startapp learning_logs

Command "startapp 'appname'" tells Django to create the infrastructure needed to create an app.
It creates it in a new folder called learning_logs, in this case.

Defining Models:

Models are what Django uses to access and manage data. They are in models.py.
Models are just classes using the Django models module, they tell Django how to work with the data.

In our case we'll need our user to create a number of topics in their learning log.
Each entry they make will be tied to a topic and these topics will be displayed as text.
We'll also need to store the timestamp for each entry, so we can show users when they made each entry.

Check models.py.
https://docs.djangoproject.com/en/2.2/ref/models/fields/

Activating Models:

To actually use our models, we need to manually tell Django to include them in the overall project.
In settings.py, in the INSTALLED_APPS list, you'll see which apps are installed and working together.
Add your apps as items in that list. (See settings.py)

Next we'll need to tell Django to modify the database so it can store information related to the models.
Run the following in console:

python manage.py makemigrations learning_logs

The makemigrations command tells Django to figure out how to store the data associated
with our new models in the database.
Will return something like the following:

Migrations for 'learning_logs':
learning_logs\migrations\0001_initial.py
- Create model Topic

It creates an initial migration file, which needs to be applied using the migrate command:
python manage.py migrate

If run correctly, it should give this as in the last two lines:
Running migrations:
Applying learning_logs.0001_initial... OK

Whenever we want to modify Learning Log, we follow these THREE STEPS:
1. Modify models.py
2. Call makemigrations on learning_logs (the appname)
3. Run the 'manage.py migrate' command to apply and modify the database

The Django Admin Site:

The Django admin site lets administrators work with their models more easily.

Setting Up a Superuser:

Django lets you set up a superuser, who has all privileges available on the site.
Run the follwing in console and respond to the prompts to create a superuser:

python manage.py createsuperuser
Username (leave blank to use 'jonat'): drudzi
Email address: jonathan.r.axelsson@gmail.com
Password: xxxxx
Password (again): xxxxx
Superuser created successfully.

Registering a Model with the Admin Site:

The admin site doesn't include all models, our own models need to be added manually.
To register your own apps to the Django admin site, go to admin.py.
Reach the admin site through localhost:port/admin

Adding topics:

Topics will be under Learning_Logs on the admin site.
We'll add Chess and Rock Climbing as topics to have some data to work with.

Defining the Entry Model:

For a user to record what they've been learning about their topics,
we need to define a model for the kinds of entries users can make in their learning logs.
Each entry needs to be associated with a certain topic.
This kind of relationship is called a many-to-one relationship,

meaning that many entries can be associated with one topic.

We place the Entry model-class in models.py.

Next, we'll migrate the Entry class like we did with the Topic class.
Also register it to the admin site in admin.py.

To get some more data to work with, we add three entries, two for chess and one for climbing.

The Django Shell:

The Django Shell is an interactive environment where we can examine our data programmatically. This environment is great for testing and troubleshooting your projects.

python manage.py shell
This command runs the Django shell.

When entered in a venv it launches a Python interpreter which allows you to explore the data stored in your project's database.

```
>>> from learning_logs.models import Topic, Entry
>>> topics = Topic.objects.all()
```
Topic.objects.all() returns a type of list called queryset of all the Topic instances:
<QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>

```
>>> for topic in topics:
...   print(topic.id, topic)
1 Chess
2 Rock Climbing
```
The for loop runs through the topics queryset, the id returns the index/id of the instance.
We can clearly see that Chess has ID of 1 and Climbing an ID of 2.

If you know the ID of an object, you can use Topic.objects.get(id=x) to examine it:

```
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
>>> topic.date_added
datetime.datetime(2019, 2, 19, 1, 55, 31, 98500, tzinfo=<UTC>)
```

You can also use ForeignKeys, connections between models.
In our case, entries are connected to a specific topic.

```
>>> topic.entry_set.all()
```
Use this formula: model(lowercase).model(lowercase)_set.all()

From the code above, we get a queryset of the entries connected to the topic.

NOTE:
Each time you modify your models, you'll need to restart the shell to see the changes.
Press CTRL-Z and ENTER to exit a shell or exit().

More on queries and Django shell:
https://docs.djangoproject.com/en/3.0/topics/db/queries/

MAKING PAGES: The Learning Log Home Page:

Making web pages with Django consists of three steps:
Defining URL patterns
Writing views
Writing templates

Order doesn't matter but we'll do it in the order above.

A URL pattern describes the way the URL is laid out.
It tells Django what to look for when matching a browser request with a site's URL...
so it knows which page to return.

Each URL maps to a certain VIEW, which is a function that
retrieves and processes the data needed for that page.

These views often render the page using a TEMPLATE, which contains the overall structure of the page.

Mapping a URL:

Users request pages by entering URLs into a browser and clicking links,
so we need to decide what URLs are needed for our site.

We include our app-URLs in the main urls.py and then define the apps' URLs in their own urls.py.
See the comments in urls.py.

Writing a View:

A view function takes in information about a request, prepares the data needed to generate
that page, and then sends the data back to the browser, often by using a template
that defines what the page will look like.

We write our index-view-function inside views.py.
See views.py.

Writing a Template:

The template defines what the page should look like,
and Django fills in the relevant data each time the page is requested.ä
A template allows you to access any data provided by the view.
Because our view-func for the home page provided no data, this template will be simple.

We'll set up a templates-folder inside the app-directory (learning_logs).
Kind of weirdly we will also set up a new folder in templates called learning_logs.
This might seem a little redundant, but it sets up a beneficial structure for Django.

After our templates folder is set up, we'll create the actual template file inside
the inner learning_logs folder and we'll name it index.html.

It's a very simple file, it looks like this:

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any
topic you're learning about.</p>
```

It's HTML code. signifies paragraphs. opens it and closes it.

Now, when you run the server and go to the base URL, you'll see this simple page using index.html template.

BUILDING ADDITIONAL PAGES:

Now that we understand and have established a routine for building pages,
we'll start building out our Learning Log.

Next we'll build two pages that both display data.
A page that lists all topics and a page that shows the entries for a certain topic.

We'll use our regular page-building routine, except that we'll create a base template,
that all templates in the project can inherit from.

Template Inheritance:

When building a website, some elements will always need to be repeated on each page.
Rather than writing these elements directly into each page,
you can write a base template containing the repeated elements
and then have each page inherit from the base template.

"This approach lets you focus on developing the unique aspects of each page,
and makes it much easier to change the overall look and feel of the project."

The Parent Template:

We'll create our base.html template in the same directory as our other templates:

```
<p>
<a href="{% url 'learning_logs:index' %}">Learning Log</a>
</p>

{% block content %}{% endblock content}
```

The HTML paragraph contains an anchor tag, which is used to set a custom name for a link.
Kind of like covering your link with another name.
The href, which holds the actual URL is a template tag, indicated by
braces and percent signs.
A template tag generates information to be displayed on a page.
{% url 'learning_logs:index' %} generates a URL matching the URL found
in learning_logs/urls.py with the name index.
The template tag can find the learning_logs namespace due to the 'app_name'
variable we defined in its urls.py.

Using a template tag like this is efficient, because it will update automatically
if we change the URL in urls.py.

Now, each page inheriting from this base template will have a link to the home page.

{% block content %}{% endblock content} is a pair of block tags.
This block, called content is a placeholder for a block of content:
the child template will define what information goes in here.
Note that child templates don't have to define every block from its parent.
You can reserve as many blocks in your parent templates as you'd like.

The Child Template:

index.html will now be a child template to base.html:

```
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>Learning Log helps you keep track of your learning, for any
  topic you're learning about.</p>
{% endblock content %}
```

{% extends "learning_logs/base.html" %} is the first line and it's the code
that tells it to inherit from the base template.
A child template must have an extends tag with a template to inherit from.
It allows the child to define the reserved blocks.

Next, we define what's in the content block using a pair of block tags.
Note that the content inside blocks should be indented, in HTML most common is 2 spaces.
Also note that endblocks don't need a block name, but it makes it readable.

Using template inheritance will make your life easier. Much so.
It simplifies each unique page and it's easier to make consistent changes.

The Topics Page:

Let's create the Topics page, which shows users what Topics they have created.

The Topics URL Pattern:

We define the Topics URL in learning_logs/urls.py and set it to 'topics'.

The Topics View:

We'll create a 'topics' function in views.py which needs to retrieve some data
from the database and send it to the template.
See learning_logs/views.py.

The Topics Template:

The topics template will recieve the context dictionary from our topics() in views.py.
We'll make the template file in templates/learning_logs and name it topics.html.

```
...
{% extends "learning_logs/base.html" %}

{% block content %}

 <p>Topics</p>

 <ul>
  {% for topic in topics %}
   <li>{{ topic }}</li>
  {% empty %}
   <li>No topics have been added yet.</li>
  {% endfor %}
 </ul>

{% endblock content %}
...
```

On the first line, we inherit the template from the base template using extends...

Then we open the content-block and close it at the end.

Later we open a UL-tag, in which everthing will be displayed as items in a bullet list.
Inisde this UL-tag, we've got another template tag ({% %}) where we're running a for-loop.
It loops through the topics-list (queryset), which our topics-view provides via the context.

Notice that this template-code differs from Python. The for-loop, which is normally
closed using indentations in Python needs an explicit {% endor %}-tag in templates.
Like this:

```
{% for item in list %}
  do something with each item
{% endfor %}
```

In the loop, we want to turn every topic into an item in the bulleted list.
The topic will be assigned to the varaible topic, and to use a variable in a template
we wrap them in double-braces, which tells Django it's a variable we want to display.
Like this:

```
{{ variable }}
```

We'll write this {{ variable }} inside an -tag, a list item.
Anything passed in between an li-tag will appear as a bulleted item if inside a UL-tag.

Then, the {% empty %} template-tag, tells Django what to do if the given list is empty.
If that's the case we'll display a message giving the user that information.

The last step for completing this page is to add a link to the topics page
in the base template.
We do it the same way as we did when adding the Learning Log home page link.
We use a {% url %} template tag inside an anchor-tag.

Individual Topic Pages:

The next step is to create a page that can focus on a single topic.
We want this page to show the name of the current topic, and its entries.
We'll also change the bulleted topics on the topics page so they link to
their corresponding individual topic page.ä

The Topic URL Pattern:

This URL will be a little different to the earlier URLs we've defined.
It will need to use the topic's id-attribute to indicate which topic was requested.

```
...
path('topics/<int:topic_id>/', views.topic, name='topic')
...
```

Let's look at the first argument above, where we define the URL.
The first part of the string (topics) tells Django to look for URLs with
the word 'topics' after the base URL.
The second part, <int:topic_id> searches for an integer which it will store in
an argument called topic_id.

When Django finds a URL request matching this pattern, it will call the associated
view-function topic() in views.py with the topic_id argument.
In the view function, we'll use the topic_id argument to get the correct
topic inside the function.

The Topic View:

We create the topic() function inside views.py.

The Topic Template:

We write the template in topic.html.

```
...
{% extends 'learning_logs/base.html' %}

{% block content %}

 <p>Topic: {{ topic }}</p>

 <p>Entries:</p>
 <ul>
 {% for entry in entries %}
  <li>
   <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
   <p>{{ entry.text|linebreaks }}</p>
  </li>
 {% empty %}
  <li>There are no entries for this topic yet.</li>
 {% endfor %}
 </ul>

{% endblock content %}
...
```

We start by extending the template from the base template, like usual.
Then we display the current topic using the topic-variable we defined in the context.

Next, we start a UL-tag, which displays items in a bullet-list.
We want to display our entries, so we create a for loop of our entries-set from the context.

The first thing we do inside the for-loop is to start a new list item tag, which will be a bullet.
Each bullet will list two pieces of information, first the timestamp and
then the full text of each entry.

In the first paragraph, we get the date_added attribute from each item in the entries set.
The vertical line afterwards indicates a template filter,
which is a function that modifies the value in a template variable.
For the timestamp, we'll use the date-filter to format the displayed output.

date:'M d, Y H:i' displays timestamps in the format January 1, 2020 15:00

For the full text we use the linebreaks filter which ensures that long text
entries include line breaks in a format understood by browsers,
rather than showing a block of uninterrupted text.

See more about filters and templates:
https://docs.djangoproject.com/en/3.0/ref/templates/builtins/

We also include something for Django to display if the entries set is empty.

Links from the Topics Page:

We need to update the topics.html template so each topic links to the appropriate page.
Instead of writing only the topic-variable for every list item, we'll use
an anchor tag and a URL-tag to generate the proper link.

```
<a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
```

In the URL template tag, we first get the URL pattern with the name topic, which we defined.
This URL-pattern also requires a topic_id argument to link to the correct topic page.
The 'topic.id' will give it that, it returns the ID of the current topic as a numerical value.

PS.
Note that there is a difference between topic.id and topic_id.
'topic.id' examines a topic and returns the numerical value of the corresponding ID.
'topic_id' is a set variable referencing to that ID in the code.
Using these expressions in the wrong way can cause error, make sure
to use them correctly.