Beginner's Python Cheat Sheet - Django

What is Django?

Django is a web framework that helps you build interactive websites using Python. With Django you define the kind of data your site will work with, and the ways your users can work with that data.

Django works well for tiny projects, and just as well for sites with millions of users.

Installing Django

It's usually best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you're working in an active virtual environment.

Create a virtual environment

```
$ python -m venv ll_env
```

Activate the environment (macOS and Linux)

```
$ source ll_env/bin/activate
```

Activate the environment (Windows)

```
> ll_env\Scripts\activate
```

Install Django to the active environment

```
(ll_env)$ pip install Django
```

Creating a project

To start we'll create a new project, create a database, and start a development server.

Create a new project

Make sure to include the dot at the end of this command.

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

Create a database

```
$ python manage.py migrate
```

View the project

After issuing this command, you can view the project at http://localhost:8000/.

```
$ python manage.py runserver
```

Create a new app

A Django project is made up of one or more apps.

```
$ python manage.py startapp learning_logs
```

Working with models

The data in a Django project is structured as a set of models. Each model is represented by a class.

Defining a model

To define the models for your app, modify the file models.py that was created in your app's folder. The `__str__()` method tells Django how to represent data objects based on this model.

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""

    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

Activating a model

To use a model the app must be added to the list `INSTALLED_APPS`, which is stored in the project's settings.py file.

```
INSTALLED_APPS = [
    # My apps.
    'learning_logs',

    # Default Django apps.
    'django.contrib.admin',
]
```

Migrating the database

The database needs to be modified to store the kind of data that the model represents. You'll need to run these commands every time you create a new model, or modify an existing model.

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

Creating a superuser

A superuser is a user account that has access to all aspects of the project.

```
$ python manage.py createsuperuser
```

Registering a model

You can register your models with Django's admin site, which makes it easier to work with the data in your project. To do this, modify the app's admin.py file. View the admin site at http://localhost:8000/admin/. You'll need to log in using a superuser account.

```
from django.contrib import admin

from .models import Topic

admin.site.register(Topic)
```

Building a simple home page

Users interact with a project through web pages, and a project's home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.

Mapping a project's URLs

The project's main urls.py file tells Django where to find the urls.py files associated with each app in the project.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('learning_logs.urls')),
]
```

Mapping an app's URLs

An app's urls.py file tells Django which view to use for each URL in the app. You'll need to make this file yourself, and save it in the app's folder.

```
from django.urls import path

from . import views

app_name = 'learning_logs'
urlpatterns = [
    # Home page.
    path('', views.index, name='index'),
]
```

Writing a simple view

A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app's views.py file. This simple view function doesn't pull in any data, but it uses the template index.html to render the home page.

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request,
            'learning_logs/index.html')
```

Online resources

The documentation for Django is available at docs.djangoproject.com/. The Django documentation is thorough and user-friendly, so check it out!

Building a simple home page (cont.)

Writing a simple template

A template sets up the structure for a page. It's a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.

The home page template will be saved as learning_logs/templates/learning_logs/index.html.

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

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

Template Inheritance

Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site.

The parent template

The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.

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

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

The child template

The child template uses the `{% extends %}` *template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.*

```
{% 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 %}
```

Template indentation

Python code is usually indented by four spaces. In templates you'll often see two spaces used for indentation, because elements tend to be nested more deeply in templates.

Another model

A new model can use an existing model. The `ForeignKey` *attribute establishes a connection between instances of the two related models. Make sure to migrate the database after adding a new model to your app.*

Defining a model with a foreign key

```
class Entry(models.Model):
    """Learning log entries for a topic."""
    topic = models.ForeignKey(Topic,
            on_delete=models.CASCADE)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return f"{self.text[:50]}..."
```

Building a page with data

Most pages in a project need to present data that's specific to the current user.

URL parameters

A URL often needs to accept a parameter telling it what data to access from the database. The URL pattern shown here looks for the ID of a specific topic and assigns it to the parameter `topic_id`.

```
urlpatterns = [
    --snip--
    # Detail page for a single topic.
    path('topics/<int:topic_id>/', views.topic,
            name='topic'),
]
```

Using data in a view

The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a `context` *dictionary to the template, containing data that should be displayed on the page. You'll need to import any model you're using.*

```
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topic.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
    }
    return render(request,
            'learning_logs/topic.html', context)
```

Restarting the development server

If you make a change to your project and the change doesn't seem to have any effect, try restarting the server:

$ python manage.py runserver

Building a page with data (cont.)

Using data in a template

The data in the view function's `context` *dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces.*

The vertical line after a template variable indicates a filter. In this case a filter called `date` *formats date objects, and the filter* `linebreaks` *renders paragraphs properly on a web page.*

```
{% 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 yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

The Django shell

You can explore the data in your project from the command line. This is helpful for developing queries and testing code snippets.

Start a shell session

```
$ python manage.py shell
```

Access data from the project

```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
>>> topic.entry_set.all()
<QuerySet [<Entry: In the opening phase...>]>
```

More cheat sheets available at

ehmatthes.github.io/pcc_2e/

Beginner's Python Cheat Sheet - Django, Part 2

Users and forms

Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and some may be public. Django's forms allow users to enter and modify their data.

User accounts

User accounts are handled by a dedicated app which we'll call `users`. Users need to be able to register, log in, and log out. Django automates much of this work for you.

Making a users app

After making the app, be sure to add `'users'` to `INSTALLED_APPS` in the project's settings.py file.

```
$ python manage.py startapp users
```

Including URLs for the users app

Add a line to the project's urls.py file so the users app's URLs are included in the project.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('', include('learning_logs.urls')),
]
```

Using forms in Django

There are a number of ways to create forms and work with them. You can use Django's defaults, or completely customize your forms. For a simple way to let users enter data based on your models, use a `ModelForm`. This creates a form that allows users to enter data that will populate the fields on a model.

The `register` view on the back of this sheet shows a simple approach to form processing. If the view doesn't receive data from a form, it responds with a blank form. If it receives `POST` data from a form, it validates the data and then saves it to the database.

User accounts (cont.)

Defining the URLs

Users will need to be able to log in, log out, and register. Make a new urls.py file in the users app folder.

```
from django.urls import path, include

from . import views

app_name = 'users'
urlpatterns = [
    # Include default auth urls.
    path('', include(
            'django.contrib.auth.urls')),

    # Registration page.
    path('register/', views.register,
            name='register'),
]
```

The login template

The login view is provided by default, but you need to provide your own login template. The template shown here displays a simple login form, and provides basic error messages. Make a templates folder in the users folder, and then make a registration folder in the templates folder. Save this file as login.html. The path should be users/templates/registration/login.html.

The tag `{% csrf_token %}` helps prevent a common type of attack with forms. The `{{ form.as_p }}` element displays the default login form in paragraph format. The `<input>` element named `next` redirects the user to the home page after a successful login.

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

{% block content %}

  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form method="post"
      action="{% url 'users:login' %}">

    {% csrf token %}
    {{ form.as_p }}
    <button name="submit">Log in</button>

    <input type="hidden" name="next"
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

User accounts (cont.)

Showing the current login status

You can modify the base.html template to show whether the user is currently logged in, and to provide a link to the login and logout pages. Django makes a `user` object available to every template, and this template takes advantage of this object.

The tag with `user.is_authenticated` allows you to serve specific content to users depending on whether they have logged in or not. The `{{ user.username }}` property allows you to greet users who have logged in. Users who haven't logged in see links to register or log in.

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

  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">
      Log out
    </a>
  {% else %}
    <a href="{% url 'users:register' %}">
      Register
    </a> -
    <a href="{% url 'users:login' %}">
      Log in
    </a>
  {% endif %}

</p>

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

The logged_out template

The default logout view renders the page using the template logged_out.html, which needs to be saved in the users/templates/registration/ folder.

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

{% block content %}
  <p>
    You have been logged out. Thank you
    for visiting!
  </p>
{% endblock content %}
```

User accounts (cont.)

The register view

The register view needs to display a blank registration form when the page is first requested, and then process completed registration forms. A successful registration logs the user in and redirects to the home page.

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """Register a new user."""

    if request.method != 'POST':
        # Display blank registration form.
        form = UserCreationForm()

    else:
        # Process completed form.
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()

            # Log in, redirect to home page.
            login(request, new_user)
            return redirect(
                    'learning_logs:index')

    # Display a blank or invalid form.
    context = {'form': form}

    return render(request,
        'registration/register.html', context)
```

Styling your project

The `django-bootstrap4` app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at django-bootstrap4.readthedocs.io/.

Deploying your project

Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment.

You'll need to install a set of Heroku command line tools, and use Git to track the state of your project. See devcenter.heroku.com/, and click on the Python link.

User accounts (cont.)

The register template

The register.html template shown here displays the registration form in paragraph format.

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

{% block content %}

  <form method='post'
      action="{% url 'users:register' %}">

    {% csrf_token %}
    {{ form.as_p }}

    <button name='submit'>Register</button>
    <input type='hidden' name='next'
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

Connecting data to users

Users will have data that belongs to them. Any model that should be connected directly to a user needs a field connecting instances of the model to a specific user.

Making a topic belong to a user

Only the highest-level data in a hierarchy needs to be directly connected to a user. To do this import the `User` model, and add it as a foreign key on the data model.

After modifying the model you'll need to migrate the database. You'll need to choose a user ID to connect each existing instance to.

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    owner = models.ForeignKey(User,
            on_delete=models.CASCADE)

    def __str__(self):
        return self.text
```

Querying data for the current user

In a view, the `request` object has a `user` attribute. You can use this attribute to query for the user's data. The `filter()` method then pulls the data that belongs to the current user.

```
topics = Topic.objects.filter(
        owner=request.user)
```

Connecting data to users (cont.)

Restricting access to logged-in users

Some pages are only relevant to registered users. The views for these pages can be protected by the `@login_required` decorator. Any view with this decorator will automatically redirect non-logged in users to an appropriate page. Here's an example views.py file.

```
from django.contrib.auth.decorators import \
        login_required
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
```

Setting the redirect URL

The `@login_required` decorator sends unauthorized users to the login page. Add the following line to your project's settings.py file so Django will know how to find your login page.

```
LOGIN_URL = 'users:login'
```

Preventing inadvertent access

Some pages serve data based on a parameter in the URL. You can check that the current user owns the requested data, and return a 404 error if they don't. Here's an example view.

```
from django.http import Http404
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --snip--
```

Using a form to edit data

If you provide some initial data, Django generates a form with the user's existing data. Users can then modify and save their data.

Creating a form with initial data

The `instance` parameter allows you to specify initial data for a form.

```
form = EntryForm(instance=entry)
```

Modifying data before saving

The argument `commit=False` allows you to make changes before writing data to the database.

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

More cheat sheets available at

ehmatthes.github.io/pcc_2e/