TRY
DJANGO
EST. 2016

Level 1 – Section 1

Getting Started

What Is Django?

What Do You Need to Know to Take This Course?

Basic Python

Try Python + Flying Through Python

Basic HTML and CSS

Front-end Foundations + Front-end Formations

Basic Database Concepts

Try SQL + The Sequel to SQL

What Is Django?

Django is an open-source MVC (or MTV) framework written in Python for building web applications.

It does this by including functionality to handle common web development tasks — right out of the box!

Others Using Django

Others Using Django

TRY DJANGO

Others Using Django

Django at a High Level

Today, most web applications including Django send data to the server to validate, process, and render HTML.

How Data Moves Through a Django App

You could do these things in one script, but separating these components makes it easier to maintain and collaborate on the project!

The Django MTV Framework

The validation, rendering, and processing is taken care of by separate components in Django: **the model, template, and view (MTV).**

If you're familiar with MVC, then you can think of the view as the controller and the template as the view, so MVC ⟶ MTV in Django.

The App We'll Create in This Course

Installing Django

Steps for installing the Django framework:

1. Make sure Python is installed. We'll be using Python 3, but Python 2 works as well.
2. Then we can use **`pip`** to install Django:

```
> pip install django

Collecting django
  Downloading Django-1.9.5-py2.py3-none-any.whl (6.6MB)
    100% |████████████████████████████████| 6.6MB 92kB/s
Installing collected packages: django
Successfully installed django-1.9.5
```

To learn more about setting up your Django environment, check out our install video:
go.codeschool.com/django-setup

Code Conventions Used in This Course

Here are all of the different places we'll be writing code in this course:

```
>    console commands
```

Using your operating system's command-line interface

```
>>>    Django shell commands
```

Using Django's interactive shell, the **>>>** symbol shows up

script_name.py

```
# Create your script here.
```

Writing code in Django script files

Django Projects vs. Apps

Let's say we have a Django project, Pinterest, that has the .com, blog, and jobs pages as separate apps in Django.

The outer project has its related settings and files.

The apps inside have their own directory with related files.

Project

Pinterest

Apps

pinterest.com

blog.pinterest.com

careers.pinterest.com

Creating a Django Project

Once Django is installed, we'll create our Django project.

```
> django-admin startproject Treasuregram
```

Our directory structure for our Django project, TreasureGram, is created:

Holds our project settings

Holds our project's URLs

Utility for administrative tasks

Now we'll run our Django server to see our project running!

Running Our Project

```
> python manage.py runserver
```

manage.py is in the top-level project directory

```
System check identified no issues (0 silenced).
April 01, 2016 - 16:03:02
Django version 1.9.4, using settings 'Treasuregram.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

127.0.0.1 is also called **localhost**, and **:8000** means **port 8000**

Tells you how to create your first app

We'll only have one app: the main .com, which we'll call **main_app** and will add next.

Creating an App Inside Our Project

Now we'll create our main app inside of our TreasureGram project:

```
> python manage.py startapp main_app
```

Our app has its own directory and generated files.

Now that we have our app, let's create our first view in **`views.py`**.

The First View We're Going to Create

A view is simply a Python function that takes in a web request and returns a web response.

For our first view, we want to return a simple text response.

Our first view will display a greeting at **`localhost:8000/index`**

Opening views.py **to Add Our View**

views.py **will hold our view functions and contains this single import statement to start.**

views.py

```
from django.shortcuts import render

# Create your views here.
```

We'll use **`render`** later to display our templates, but for now we'll just return a simple **`HttpResponse`**.

Importing HttpResponse

We're going to return a simple text HTTP response, so we need to import the **`HttpResponse`** class from the **`django.http`** module.

views.py

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

# Create your views here.
```

To import a specific class from a module, you can type:

`from` ***`package.module`*** `import` ***`class (or function)`***

Creating Our Index View

A view is a function that takes a web request and returns a web response.

views.py

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

# Create your views here.
def index(request):
    return HttpResponse('<h1>Hello Explorers!</h1>')
```

The functions in **`views.py`** are called **views** in Django, and they all take in a request.

This is the simplest type of view in Django.

But to call it, we'll need to map it to a URL!

The URLs Dispatcher

We want the URL server/index/ to go to our index view that returns, “Hello Explorers!”

Creating the New URL in the URL Dispatcher

The project's URL dispatcher is in urls.py and will send the URL to the matching view.

Treasuregram/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from main_app import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    # localhost/index
    url(r'^index/',
        views.index),
]
```

We need to import our app's views to call our index view

r means the string is raw and won't be escaped

This is a regular expression (regex) pattern.

Note: If you want to learn more regex, check out our Breaking the Ice With Regular Expressions course!

Our New URL Pattern Will Go to Our Index View

Treasuregram/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from main_app import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    # localhost/index
    url(r'^index/',
        views.index),
]
```

main_app/views.py

```
...

# Create your views here.
def index(request):
    return HttpResponse('Hello…')
```

If the user visits **localhost/index**, they'll be sent to **main_app**'s index view, which will return the **HttpResponse**.

The 2nd parameter means:

Look inside the **views.py** file

Call the index view function

Note: This is the same Python path we've seen before — **module.function** — where views is the module, and index is the function.

Seeing Our First View

Our Django server should be still running, so we can refresh and see our new view at localhost:8000/index/.

TRY
DJANGO
EST. 2016

Level 1 – Section 2

Getting Started

URL Dispatcher Best Practices

Shortening Our URL

Right now localhost:8000/index/ goes to our homepage.

Ugly URL

When we eventually host our site, we wouldn't want google.com/index/ to be our homepage — we would want it to be just google.com.

We'd like to be able to just type **`localhost:8000/`**

Refactoring the Project's URLs Dispatcher

First, we'll remove the `index/` from our regex and match an empty path to load the index view.

Treasuregram/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from main_app import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    # localhost:8000 will go here
    url(r'^',
      views.index),
]
```

This regex will match an empty path.

Best Practice: The App URLs Dispatcher

It's a best practice to have a project URL dispatcher ***and*** an app URL dispatcher.

What we need to create

a little weird how the audio cuts out here, possibly need to say something about creating that now?

Refactoring the Project's URLs Dispatcher

The project's URL dispatcher is in urls.py and will send the URL to the matching app.

Import **include**, so we can include **main_app**'s URLs

Treasuregram/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    # localhost:8000 will go here
    url(r'^',
        include('main_app.urls')),
]
```

If this pattern is matched, then the rest of the URL will be matched by **main_app**'s **urls.py**.

We have URL patterns in our base TreasureGram directory, but we also want to create one in each app.

Create a new **urls.py** in the **main_app** directory.

Creating the App's URLs Dispatcher

We'll still **import url** and we'll also **import** our app's **views**.

main_app/urls.py

```
from django.conf.urls import url
from . import views
```

If you want to import a specific module from your current app, you can leave off the package and type the following:

`from . import` ***module***

Creating the App's URLs Dispatcher

Then the app-specific URL dispatcher can handle any URLs that have an empty path.

main_app/urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.index),
]
```

This pattern checks that the URL has an empty path, which will go to the homepage.

The $ terminates the regex.

Our View Still Works After Refactoring

Now `localhost:8000/` goes to our homepage.

TRY
DJANGO
EST. 2016

Level 2 – Section 1

Templates

The Template of Data

The Final Homepage

Now we're going to build the homepage, which shows a list our currently discovered treasures.

The URL-View-Template Process

localhost:8000/

URL Dispatcher

Matches the requested URL to the correct view

index()

View

Collects needed data and renders templates

render index.html

Template

Defines the HTML to be rendered

We need to create the HTML file to be rendered, since we already have the URL and view.

Creating the Templates Directory

We need to create a templates directory to store our templates in our app.

Create a templates directory for our templates

Django automatically looks for templates in directories named "templates" in your app, if your app is registered with the Django project in settings.

Registering Your App in Settings

We need to add our main_app app to our INSTALLED_APPS so Django will know where to find things like its templates folder.

settings.py

```
INSTALLED_APPS = [
    'main_app',

    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Our **main_app** app

All of the apps that came pre-installed with Django

Our First Template index.html

Let's create a basic HTML page in our templates directory that displays the title TreasureGram.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>TreasureGram</title>
  </head>
  <body>
    <h1>TreasureGram</h1>
  </body>
</html>
```

This template will render like this

TreasureGram

Making the View Render an HTML Template

views.py

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

def index(request):
    return render(request, 'index.html')
```

Remember this **import** statement from the starting file in level 1? We'll use it now.

We can delete this now since we'll be using **render**.

Call the **render** function with the **request** and template name as parameters to render a template.

Rendered index.html

If we inspect the page, we can see the HTML we wrote in our template.

How Do We Get Dynamic Data in Our Template?

Now we want to pass dynamic data to our template.

How can we pass data like this to our template?

Passing Dynamic Data to the Template

We can use a context to pass data from a view to a template.

views.py

```
from django.shortcuts import render

def index(request):
    name = 'Gold Nugget'
    value = 1000.00
    context = {'treasure_name': name,
               'treasure_val': value }

    return render(request, 'index.html', context)
```

A **context** is a dictionary that maps template variable names to Python objects.

We can pass the **context** as a third parameter to our **`render()`** function.

TRY DJANGO

Django Template Language

In the template, you can access a variable from the context by surrounding it with {{ }}.

Called mustaches

```
context =
{'treasure_name': name,'treasure_val': value }
```

We can access the values in the context by their **keys**.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>TreasureGram</title>
  </head>
  <body>
    <h1>TreasureGram</h1>
    <p>{{ treasure_name }}</p>
    <p>{{ treasure_val }}</p>
  </body>
</html>
```

During rendering, all of the **{{ variables }}** are replaced with their values, which are looked up in the context.

Rendered Template With Dynamic Data

Now we see our template rendered with the values of treasure_name and treasure_val!

Our passed in variables:
`treasure_name`
`treasure_val`

TRY
DJANGO
EST. 2016

Level 2 – Section 2

Templates

More Template Fun

Passing More Data in a Dictionary Context

The context dictionary worked for simply sending two values...

But what if we have more complex data? Or a bunch of treasures?

Use a class to define information about a `Treasure` object.

Use a list to store all of the `Treasure` objects — called **treasures**.

Then we have just **one** variable to put in our **context** — **treasures**.

Creating Our Treasure Class With a Constructor

The Treasure class will have a constructor that sets all of its attributes.

```
class Treasure:
    def __init__(self, name, value, material, location):
        self.name = name
        self.value = value
        self.material = material
        self.location = location
```

Then we can create a **Treasure** object and set its attributes with one line of code!

```
t1 = Treasure('Gold Nugget', 500.00, 'gold', "Curly's Creek, NM")
t2 = Treasure("Fool's Gold", 0, 'pyrite', "Fool's Falls, CO")
```

This creates two **Treasures**, each with its own values (or instance variables).

Treasure:
- Gold Nugget
- 1000
- gold
- Curly's...

Treasure:
- Fool's Gold
- 15
- pyrite
- Fool's...

TRY DJANGO

Adding the Treasure Class to views.py

We'll then add our Treasure class to views.py and add a list of Treasure objects.

views.py

```
...

class Treasure:
    def __init__(self, name, value, material, location):
        self.name = name
        self.value = value
        self.material = material
        self.location = location

treasures = [
    Treasure('Gold Nugget', 500.00, 'gold', "Curly's Creek, NM")
    Treasure("Fool's Gold", 0, 'pyrite', "Fool's Falls, CO")
    Treasure('Coffee Can', 20.00, 'tin', 'Acme, CA')
]
```

At the bottom of **views.py**, we will add a **Treasure** class to store these attributes.

Then, we can create a list of all of our treasures (to create our context).

Note: The **Treasure** class will be replaced with our database model later.

Creating a Context With a List of Objects

Then, we can pass our list of treasures as our context in our `index` view.

views.py

```
from django.shortcuts import render

def index(request):
    return render(request, 'index.html', {'treasures':treasures})

class Treasure:
    def __init__(self, name, value, material, location):
        self.name = name
        self.value = value
        self.material = material
        self.location = location

treasures = [
    Treasure('Gold Nugget', 500.00, 'gold', "Curly's Creek, NM")
    Treasure("Fool's Gold", 0, 'pyrite', "Fool's Falls, CO")
    Treasure('Coffee Can', 20.00, 'tin', 'Acme, CA')
]
```

We put the context in directly instead of creating a context variable first.

Django Template Language Tags

Tags provide Python-like logic for the rendering process. A tag starts with {% and ends with %}.

We can create a **for** loop similar to Python code using tags.

```
{% for treasure in treasures %}
    <p>{{ treasure.name }}</p>
    <p>{{ treasure.value }}</p>
{% endfor %}
```

The HTML we want to generate goes in the loop.

The **endfor** tag tells Django where the **for** block ends.

Using Tags in index.html

Now we'll move our for loop into the template to generate HTML for each Treasure object.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>TreasureGram</title>
  </head>
  <body>
    <h1>TreasureGram</h1>
    {% for treasure in treasures %}
      <p>{{ treasure.name }}</p>
      <p>{{ treasure.value }}</p>
    {% endfor %}
  </body>
</html>
```

Rendered Template

We can now see our list of Treasure objects!

We actually want to replace zeroes with "Unknown," since that means we don't know the value.

Adding if, else **Tags in** index.html

If tags start with {% if conditional %} and end with {% endif %}, else and elif statements can go in between.

index.html

```
...
<body>
   <h1>TreasureGram</h1>
   {% for treasure in treasures %}
     <p>{{ treasure.name }}</p>
     {% if treasure.value > 0 %}
        <p>{{ treasure.value }}</p>
     {% else %}
        <p>Unknown</p>
     {% endif %}
   {% endfor %}
</body>
</html>
```

if tag

else tag

endif tag

If the value is zero, the **`treasure.value`** is displayed.

Otherwise, "Unknown" is displayed.

There are many more Django template tags, which you can find in the docs here: go.codeschool.com/django-template-tags

Rendered Template

Now we have the building blocks for our page, but it needs some style...

If we look at our rendered template, we see the value of Fool's Gold is "Unknown"!

TRY
DJANGO
EST. 2016

Level 2 – Section 3

Templates

Styling Templates

The Styles We Want to Add to Our Page

Before

After

Adding Static Files

Let's add a CSS file to make our page look better.

style.css

```
h1 {
    color: green;
}
```

Create a static directory for our static files, like CSS, JavaScript, and static images.

Loading Static Files in Templates

Using the tag {% load staticfiles %} will make our static files directory accessible to this template.

load tag

index.html

```
{% load staticfiles %}

<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" type="text/css"
        href="{% static 'style.css' %}" />
  <title>TreasureGram</title>
</head>
<body>
...
```

This tells Django to load the **style.css** file from the static folder.

Rendered Template With CSS

We can now see our style.css file in action.

H1 is now green as defined in `style.css`.

Adding Bootstrap

We can even add a CSS framework like Bootstrap to do some style work for us.

index.html

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
  <head>
    <link href="{% static 'bootstrap.min.css' %}"
          rel="stylesheet">
    <link href="{% static 'style.css' %}"
          rel="stylesheet">
    <title>TreasureGram</title>
  </head>
  <body>
...
```

We're just going to add the minified **bootstrap.min.css** in addition to our **style.css**.

Download Bootstrap:
go.codeschool.com/bootstrap-download

Adding Static Images

We'll also add a static/images directory to hold some images, like our logo, to style our site.

Adding Bootstrap Elements to Our Template

index.html

```
...
<body>
  <nav class="navbar navbar-default navbar-static-top text-center">
    <a href="/">
      <img src="{% static 'images/logo.png' %}" alt="TreasureGram">
    </a> </nav>
<main ...></main>
</body>
...
```

Creating a navbar

Adding a static image for our logo

The resulting navbar with logo image

Adding Bootstrap Elements to Our Template

index.html

```
...
<nav ...</nav>
<main class="container" role="main">
  <div class="treasure panel panel-default">
  {% for treasure in treasures %}
  <div class="panel-heading">
    <h2 class="panel-title">{{ treasure.name }}</h2>
  </div>
  <div class="panel-body">
    {{ treasure.material }}
    {% if treasure.value > 0 %}
      {{ treasure.value }}
    {% else %}
      {{ Unknown }}
    {% endif %}
  </div>
  {% endfor %}
</div>
</main>...
```

Creating a container

Creating a panel

Note: Find out more in our Blasting Off With Bootstrap course!

Our Current Page Styled With Bootstrap

Things are looking good, but we want to organize these attributes into a table.

Adding a Table to Our Template

index.html

```
...
<main class="container" role="main">
  <div class="treasure panel panel-default">
    {% for treasure in treasures %}
    <div class="panel-heading">
      <h2 class="panel-title">{{ treasure.name }}</h2>
    </div>
    <div class="panel-body">
      <table class="treasure-table">
        <tr>
          <th>
            <img src="{% static 'images/materials-icon.png' %}" >
            Material:
          </th>
          <td>{{ treasure.material }}</td>
        </tr>
        <tr>
        ...
      </div></main>...
```

Adding a table

Adding other table rows for location and value

Rendered With Bootstrap

Now our site looks better already by adding some Bootstrap elements!

The site could be even better if we had some images for each item!

Adding an Image URL to our Treasure Class

views.py

```
from django.shortcuts import render

def index(request):
    return render(request, 'index.html', {'treasures':treasures})

class Treasure:
    def __init__(self, name, value, material, location, img_url):
        self.name = name
        self.value = value
        self.material = material
        self.location = location
        self.img_url = img_url

treasures = [
    Treasure('Gold Nugget' ..., 'example.com/nugget.jpg'),
    Treasure("Fool's Gold" ..., 'example.com/fools-gold.jpg'),
    Treasure('Coffee Can' ..., 'example.com/coffee-can.jpg')
]
```

Define and set a new attribute to store the image URL

Add the actual URLs when you create the object

Adding an img_url to Our Template

index.html

```
...
<main class="container" role="main">
  <div class="treasure panel panel-default">
    {% for treasure in treasures %}
    <div class="panel-heading">
      <h2 class="panel-title">{{ treasure.name }}</h2>
    </div>
    <div class="panel-body">

    <div class="treasure-photo">
      <img src="{{ treasure.img_url }}" height="200">
    </div>

     <table class="treasure-table">
       ...
     </table>
   </div>
   {% endfor %}
</div></main>...
```

Use the **img_url** value in an **<img>** tag in the template

The Final Index List Page

TreasureGram

Gold nugget

Material: Gold

Value: 500.00

Location: Curly's Creek, NM

Fool's gold

Material: Pyrite

Value: Unknown

Location: Curly's Creek, NM

TRY
DJANGO
EST. 2016

Level 3 – Section 1

Models

The Missing Model

Adding the Model to Our Django Flow

The Model Should Be Separate From the View

Right now, we're storing our treasures in a list, but there isn't a good interface for adding new ones.

views.py

```
...

class Treasure:
    def __init__(self, name, val, mat, loc, img_url):
        self.name = name
        self.value = val
        self.material = mat
        self.location = loc
        self.img_url = img_url

treasures = [
    Treasure('Gold Nugget', 1000.00, 'gold', 'Jackson…', …),
    Treasure('Arrow Heads', 1000000.00, 'stone', 'SLC…', …),
    ... ]
```

Currently, to add to our treasures list, we're directly editing **`views.py`**.

We want a way to edit treasures directly from the website and create something that's easy to maintain over time… A model is how you do this in Django!

TRY DJANGO

Creating Our Treasure Model

We'll create a Treasure model with the same attributes as our Treasure class, but using the steps below:

models.py

```
from django.db import models

class Treasure(models.Model):

  name = models.CharField()
  value = models.DecimalField()
  material = models.CharField()
  location = models.CharField()
  img_url = models.CharField()
...
```

1. Import the **models** class.
2. Create a class that inherits from **models.Model** so that Django knows we're creating a model.
3. Use special model types that correspond to database types.

The Old Treasure Class vs. Our Treasure Model

You can see the differences between our old Treasure class and our new model.

views.py

```
...

class Treasure:
  def __init__(...):
    self.name = name
    self.value = value
    self.material = material
    self.location = location
    self.img_url = img_url
...
```

models.py

```
from django.db import models

class Treasure(models.Model):

  name = models.CharField()
  value = models.DecimalField()
  material = models.CharField()
  location = models.CharField()
  img_url = models.CharField()
...
```

Notice the **Treasure** model doesn't need a constructor.
The model has one built-in where you use the attribute names to set each one, like so:

```
t = Treasure(name='Coffee can', value=20.00, location='Acme,CA',
             material='tin', img_url = 'example.com/coffee.jpg')
```

TRY DJANGO

Django Model Field Types

We need to use special model field types that correspond to database types:

Django	Python	SQL
`models.CharField()`	string	VARCHAR
`models.IntegerField()`	int	INTEGER (different types)
`models.FloatField()`	float	FLOAT
`models.DecimalField()`	decimal	DECIMAL

There are many more Django model field types, which you can find in the docs here: go.codeschool.com/django-field-types

TRY DJANGO

Updating Our Treasure Model

We'll define some rules for our model attributes, otherwise we would get errors like these:

```
ERRORS:
main_app.Treasure.name: (fields.E120) CharFields must define a 'max_length' attribute.
main_app.Treasure.value: (fields.E130) DecimalFields must define a 'decimal_places' attribute.
main_app.Treasure.value: (fields.E132) DecimalFields must define a 'max_digits' attribute.
```

models.py

```python
from django.db import models

class Treasure(models.Model):
    name = models.CharField(max_length=100)
    value = models.DecimalField(max_digits=10,
                                decimal_places=2)

    material = models.CharField(max_length=100)
    location = models.CharField(max_length=100)
    img_url = models.CharField(max_length=100)
```

A model describes how we want to store the data in a database, but do we need to write any SQL?

If you're familiar with databases, you usually want to set the **max_length** of your fields or give default values.

TRY DJANGO

A Model Is a Mapping to a Database Table

When we create a model and model objects in Django, we're actually creating a database table and database entries... but we don't have to write any SQL!

Python code → **Django ORM** (Object Relational Mapping) → SQL commands

The ORM translates Python code to SQL for us behind the scenes.

There's an extra step called **migrations** that needs to happen only right after you create a new model or update an existing model.

How to Perform a Migration

Let's perform a migration so we can start adding Treasure objects to our model.

1. `> python manage.py makemigrations` Makes a migration file

2. `> python manage.py migrate` Applies the migration to the database

Separate steps let you review the migration before you actually run **`migrate`**.

Migrations are like a version control system for your database.

TRY DJANGO

Let’s Make a Database Migration

We already created our Treasure model, so now we need to create a migration.

```
> python manage.py makemigrations

Migrations for 'main_app':
  0001_initial.py:
    - Create model Treasure
```

Outputs the migration file:
migrations/0001_initial.py

We Can Preview the SQL in Our Migration

Before running migrate, we can preview the SQL commands in our migration — and you can see the ORM is actually generating SQL commands behind the scenes!

```
> python manage.py sqlmigrate main_app 0001

BEGIN;
--
-- Create model Treasure
--
CREATE TABLE "main_app_treasure" ("id" integer NOT NULL
PRIMARY KEY AUTOINCREMENT, "name" varachar(100) NOT
NULL, "value" decimal NOT NULL, "material" varchar(100)
NOT NULL, "location varchar(100) NOT NULL;

COMMIT;
```

Note: You don't need to do this! ... But this can be useful if you want to verify your Python code is doing what you expect.

Let's Migrate Our Migration

We haven't run migrate yet, so there's a lot of migrations that run with built-in Django components.

```
> python manage.py migrate

Operations to perform:
  Apply all migrations: treasuregram, sessions, admin,
auth, contenttypes
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying sessions.0001_initial... OK
  Applying main_app.0001_initial… OK
```

Other Django components

main_app related

Are We Up to Date?

If you try running make migrations now, you'll see there are no changes detected.

```
> python manage.py makemigrations

No changes detected
```

And if you try to migrate now, you'll see there are no migrations to apply.

```
> python manage.py migrate

Operations to perform:
  Apply all migrations: main_app, sessions, admin,
auth, contenttypes
Running migrations:
  No migrations to apply.
```

The Django Interactive Shell

Now that we've done our migration, we can play with Treasure objects in the shell!

```
>   python manage.py shell

Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03) ...
Type "help", "copyright", "credits" or "license" for more info.
(InteractiveConsole)

>>>  from main_app.models import Treasure

>>>  Treasure.objects.all()

[]
```

The result is an empty list since we haven't added any **Treasure** objects yet...

Retrieving Objects With a QuerySet

In Django, a QuerySet represents a collection of objects from your database.

QuerySet equates to a `SELECT` statement in SQL

```
Treasure.objects
```

All QuerySets start like this

Retrieving all objects, like `SELECT * FROM Treasure`

```
Treasure.objects.all()
```

Would return a list of **Treasure** objects

Different filters can be used like limiting clauses in `SQL WHERE` or `LIMIT`

```
Treasure.objects.filter(location = 'Orlando, FL')
```

All of the **Treasure** objects located in Orlando

If you know there is only one element matching your query, you can use get()

```
Treasure.objects.get(pk = 1)
```

pk is primary key and is unique, so this will only return the **Treasure** object with the matching primary key

Creating Objects in the Shell

Now that we've done our migration, we can create Treasure objects! We'll do this in the Django shell.

```
>>> t = Treasure(name='Coffee can', value=20.00, location='Acme,
                 CA', material='tin', img_url = '…')
>>> Treasure.objects.all()
[]
>>> t.save()
>>> Treasure.objects.all()
[<Treasure: Treasure object>]
```

We need to save the object to the database before it will show up in queries.

This would be even better if it described the treasures in the list.

Adding a Descriptive __str__ method

models.py

```
from django.db import models

# Create your models here.
class Treasure(models.Model):
    name = models.CharField(max_length=100)
    value = models.DecimalField(max_digits=10,decimal_places=2)
    material = models.CharField(max_length=100)
    location = models.CharField(max_length=100)
    img_url = models.CharField(max_length=100)

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

Now the treasure's name will be shown whenever we output that object in the interactive shell.

```
>>> Treasure.objects.all()
[Coffee can]
```

More helpful output!

Creating More Treasures in the Shell

```
>>> t = Treasure(name = 'Gold Nugget', value = 500, material = 'gold',
                 location = "Curly's Creek, NM", img_url = '…')
>>> t.save()
```

```
>>> t = Treasure(name = "Fool's Gold", value = 0, material ='pyrite',
                 location = "Fool's Falls, CA", img_url = '…')
>>> t.save()
```

```
>>> Treasure.objects.all()

[Coffee can, Gold Nugget, Fool's Gold]
```

We have all of the treasures from our original list.

Updating Our Existing Code to Use Our Model

Now we need to use a QuerySet to get all of Treasure's objects in our index view.

views.py

```
from django.shortcuts import render
from .models import Treasure

def index(request):
    # Here we need to get all of Treasure's objects
    return render(request, 'index.html', {'treasures': treasures})
```

We need to import our model to use it.

If you want to import a specific class from your current app, you can leave off the package and type the following:

from ***.module*** import ***class (or function)***

Using a QuerySet to Get the Treasure Objects

We'll use Django QuerySets to get all of the Treasure objects from our model.

views.py

```
from django.shortcuts import render
from .models import Treasure

def index(request):
    treasures = Treasure.objects.all()
    return render(request, 'index.html', {'treasures': treasures})
```

Demo of Our Model in Action

Now our app should work exactly as it has been, but we know there's a new model in place behind the scenes!

TRY
DJANGO
EST. 2016

Level 3 – Section 2

Models

The Admin

The Built-in Django Admin

Creating a Super User to Use the Admin

To use the admin site, we need to create a super user.

```
> python manage.py createsuperuser

Username: sbuchanan
Email address: sbuchanan@codeschool.com
Password: ...
Password (again): ...
Super user created successfully.
```

Using the Admin

When we log in, we see our automatically generated Groups and Users, but not our `Treasure` model.

Registering Models With the Admin

In order to see our models in the admin, we need to register them in admin.py.

admin.py

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

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

Creating More Treasure Objects in the Admin

Demo of Our Updated Model in Action

Now our app has the new items we added in the admin!

TRY
DJANGO
EST. 2016