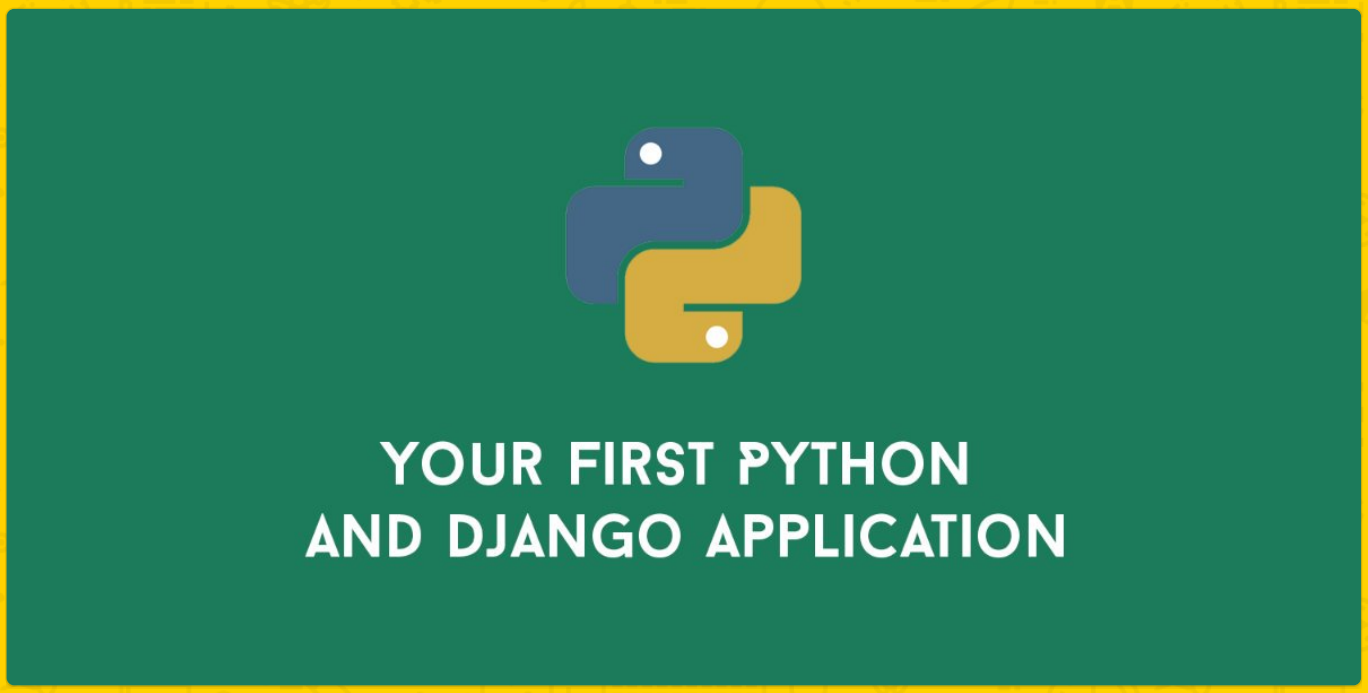
Build Your First Python and Django Application

Amos Omondi (@amos_omondi)

July 12, 2016
#python #django



You may have heard of Python before, especially if you have been coding for a while.

If not, Python is a high level, general purpose programming language. What this means is that you can use it to code up anything from a simple game to a website supporting millions of users per month.

In fact, several high profile sites with millions of visitors per month rely on Python for some of their services. Examples include YouTube and Dropbox

Table of Contents

That being said, why should you use Python in the first place? Why not one of the many other popular languages out in the wild like Ruby or PHP? Well, with Python you get the following awesome benefits:

1. Easily readable syntax.
2. Awesome community around the language.
3. Easy to learn.
4. Python is useful for a myriad of tasks from basic shell sripting to advanced web development.

When Not to Use Python

While you can easily write a Desktop app with Python using tools likewxPython, you generally would do better to use the specialized tools offered by the platform you are targeting for example .NET on Windows.

You should also not use Python when your particular use case has very specialized requirements which are better met by other languages. An example is when you are building an embedded system, a domain in which languages like C, C++ and Java dominate.

Python 2 vs Python 3

Python 2.7.x and 3.x are both being used extensively in the wild. Python 3 introduced changes into the language which required applications written in Python 2 to be rewritten in order to work with the Python 3.x branch. However, most libraries you will require to use have now been ported to Python 3.

This tutorial will use Python 3 which is at version 3.5.1 at the time of writing. The principles remain the same though and only minor syntax modifications will be required to get the code running under Python 2.7.x.

Some Python Code Samples

Hello World

As I said before, one of Python's main benefits is its' very easily readable syntax. How easy? Check out Python's version of the ubiquitous `Hello World` .

Related Course: Getting Started with JavaScript for Web Development

```
# This line of code will print "Hello, World!" to your terminal
print("Hello, World!")
```

This code prints out `Hello, World!` to the console. You can easily try out this code by visiting this site, pasting the code samples in the editor on the right side of the page, and clicking the `run` button above the page to see the output.

Conditional Logic

Conditional logic is just as easy. Here is some code to check if a user's age is above 18, and if it is, to print `Access allowed` or `Access not allowed` otherwise.

```
# read in age
age = int(input("What's your age?"))

if age >= 18:
    print("Access allowed")
elif age < 18 and age > 0:
    print("Access not allowed")
else:
    print("Invalid age")
```

The `input()` function is used to read in keyboard input. You will therefore need to type something in the terminal prompt after running the script for rest of the script to execute. Note that the `input()` function is wrapped in the `int()` function.

This is because `input()` reads in values as `strings` and yet we need age to be an `integer`. We therefore have to cast the keyboard input into a string or else we will get an error for example when checking if the string is greater than 18.

Finally, note the `else` statement which executes for any other input which doesn't fit the criteria being checked for in the if statements.

Abstract Data Types

Python also has some excellent built in abstract data types for holding collections of items. An example is a list which can be used to hold variables of any type. The following code shows how to create a list and iterate through it to print each item to the terminal.

```
# create a list called my_list
my_list = [1, 2, 3, "python", 67,  [4, 5]]

# go through my_list and print every item
for item in my_list:
    print item
```

The code above creates a list with numbers, a string and a list (yes, lists can contain other lists!). To iterate through lists, a `for-in` loop comes in handy. Remember, lists are zero-indexed so we can also access list items using indexes. For example, to output the string `python`, you can write:

```
# create a list called my_list
my_list = [1, 2, 3, "python", 67,  [4, 5]]

print(my_list[3])
```

Dictionaries

Another excellent data type Python offers out of the box is dictionaries. Dictionaries store key-value pairs, kind of like JSON objects. Creating a dictionary is quite simple as well.

```
# create a dictionary
person = {
        "name": "Amos",
        "age": 23,
        "hobbies": ["Travelling", "Swimming", "Coding", "Music"]
    }

# iterate through the dict and print the keys
for key in person:
    print(key)

# iterate through the dict's keys and print their values
for key in person:
    print(person[key])
```

Now that you know a little bit of Python, let's talk about Django.

Django

Django is a Python web framework. It is free and open source and has been around since 2005. It is very mature and comes with excellent documentation and awesome features included by default. Some excellent tools it provides are:

1. Excellent lightweight server for development and testing.
2. Good templating language.
3. Security features like CSRF included right out of the box.

There are a myriad of other useful things included in Django but you shall probably discover them as you go along. We are going to be using Django to build our website in this tutorial.

Setting Up

In this tutorial, I will show you how to get a Django website up and running. Before we get there though, first grab a copy of the latest Python from the Python website.

> Note that if you are on OSX and you have Homebrew installed you can do
>
> ```
> brew install python3
> ```
>
> After that go straight to the Getting started with Django section

After installing the correct version for your OS, you will need to make sure it is set up correctly. Open a terminal and type:

```
python3
```

You should see something resembling the following:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This is the interactive Python shell. Hit `CTRL + D` to exit it for now

Setting Up the Environment

To avoid polluting our global scope with unecessary packages, we are going to use a virtual environment to store our packages. One excellent virtual environment manager available for free is `virtualenv`. We will be using Python's package manager `pip` to install this and other packages like Django which we will require later on. First, let's get `virtualenv` installed.

```
pip install virtualenv
```

Once that is done, create a folder called projects anywhere you like then `cd` into it.

```
mkdir projects
```

```
cd projects
```

Once inside the projects folder, create another folder called hello. This folder will hold our app.

```
mkdir hello
```

At this point, we need to create the environment to hold our requirements. We will do this inside the `hello` folder.

```
virtualenv -p /usr/local/bin/python3 env
```

The `-p` switch tells virtualenv the path to the python version you want to use. Feel free to switch out the path after it with your own Python installation path. The name `env` is the environment name. You can also change it to something else which fits the name of your project.

Once that is done, you should have a folder called `env` inside your `hello` folder. Your structure should now look something like this.

```
projects
├─hello
|   ├── env
```

You are now ready to activate the environment and start coding!

```
source env/bin/activate
```

You will see a prompt with the environment name. That means the environment is active.

```
(env)
```

Installing Django

This is a simple pip install. The latest Django version at the time of writing is Django 1.9.6

```
pip install django
```

Creating an App

Now that Django is installed, we can use its start script to create a skeleton project. This is as simple as using its admin script in the following way.

```
django-admin startproject helloapp
```

Running this command creates a skeleton django app with the following structure:

```
helloapp
├─helloapp
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

When you look into the `helloapp` folder that was created, you will find a file called `manage.py` and another folder called `helloapp`. This is your main project folder and contains the project's settings in a file called `settings.py` and the routes in your project in the file called `urls.py`. Feel free to open up the `settings.py` file to familiarize yourself with its contents.

Ready to move on? Excellent.

Changing App Settings

Let's change a few settings. Open up the `settings.py` file in your favorite editor. Find a section called Installed Apps which looks something like this.

```
# helloapp/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Django operates on the concept of apps. An app is a self contained unit of code which can be executed on its own. An app can do many things such as serve a webpage on the browser or handle user authentication or anything else you can think of. Django comes with some default apps preinstalled such as the authentication and session manager apps. Any apps we will create or third-party apps we will need will be added at the bottom of the `Installed Apps` list after the default apps installed.

Before we create a custom app, let's change the application timezone. Django uses the `tz database` timezones, a list of which can be found here.

The timezone setting looks like this.

```
# helloapp/settings.py
TIME_ZONE = 'UTC'
```

Change it to something resembling this as appropriate for your timezone.

```
# helloapp/settings.py
TIME_ZONE = 'America/Los_Angeles'
```

Creating your own app

It is important to note that Django apps follow the Model, View, Template paradigm. In a nutshell, the app gets data from a model, the view does something to the data and then renders a template containing the processed information. As such, Django templates correspond to views in traditional MVC and Django views can be likened to the controllers found in traditional MVC.

That being said, let's create an app. `cd` into the first `helloapp` folder and type;

```
python manage.py startapp howdy
```

Running this command will create an app called howdy. Your file structure should now look something like this.

```
helloapp
├── helloapp
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── howdy
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   ├── models.py
│   ├── tests.py
│   └── views.py
└── manage.py
```

To get Django to recognize our brand new app, we need to add the app name to the `Installed Apps` list in our `settings.py` file.

```
# helloapp/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'howdy'
]
```

Once that is done, let's run our server and see what will be output. We mentioned that Django comes with a built in lightweight web server which, while useful during development, should never be used in production. Run the server as follows:

```
python manage.py runserver
```

Your output should resemble the following:

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

June 04, 2016 - 07:42:08
Django version 1.9.6, using settings 'helloapp.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

If you look carefully, you will see a warning that you have unapplied migrations. Ignore that for now. Go to your browser and access `http://127.0.0.1:8000/` . If all is running smoothly, you should see the Django welcome page.

We are going to replace this page with our own template. But first, let's talk migrations.

Migrations

Migrations make it easy for you to change your database schema (model) without having to lose any data. Any time you create a new database model, running migrations will update your database tables to use the new schema without you having to lose any data or go through the tedious process of dropping and recreating the database yourself.

Django comes with some migrations already created for its default apps. If your server is still running, stop it by hitting `CTRL + C` . Apply the migrations by typing:

```
python manage.py migrate
```

If successful, you will see an output resembling this one.

```
Operations to perform:
  Apply all migrations: sessions, auth, contenttypes, admin
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
```

Running the server now will not show any warnings.

Urls & Templates

When we ran the server, the default Django page was shown. We need Django to access our `howdy` app when someone goes to the home page URL which is `/`. For that, we need to define a URL which will tell Django where to look for the homepage template.

Open up the `urls.py` file inside the inner `helloapp` folder. It should look like this.

```
# helloapp/urls.py
"""helloapp URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:

https://docs.djangoproject.com/en/1.9/topics/http/urls/

Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/', include('blog.urls'))
"""
from django.conf.urls import url
from django.contrib import admin

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

As you can see, there is an existing URL pattern for the Django admin site which comes by default with Django. Let's add our own url to point to our howdy app. Edit the file to look like this.

```
# helloapp/urls.py
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^', include('howdy.urls')),
]
```

Note that we have added an import for `include` from django.conf.urls and added a url pattern for an empty

route. When someone accesses the homepage, (in our case http://localhost:8000), Django will look for more url definitions in the `howdy` app. Since there are none, running the app will produce a huge stack trace due to an `ImportError`.

```
.
.
ImportError: No module named 'howdy.urls'
```

Let's fix that. Go to the `howdy` app folder and create a file called `urls.py`. The `howdy` app folder should now look like this.

```
├── howdy
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   ├── __init__.py
│   ├── models.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
```

Inside the new `urls.py` file, write this.

```
# howdy/urls.py
from django.conf.urls import url
from howdy import views

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

This code imports the views from our `howdy` app and expects a view called `HomePageView` to be defined. Since we don't have one, open the `views.py` file in the `howdy` app and write this code.

```
 # howdy/views.py
from django.shortcuts import render
from django.views.generic import TemplateView

# Create your views here.
class HomePageView(TemplateView):
    def get(self, request, **kwargs):
        return render(request, 'index.html', context=None)
```

This file defines a view called `HomePageView` . Django views take in a `request` and return a `response` . In our case, the method `get` expects a HTTP GET request to the url defined in our `urls.py` file. On a side note, we could rename our method to `post` to handle HTTP POST requests.

Once a HTTP GET request has been received, the method renders a template called `index.html` which is just a normal HTML file which could have special Django template tags written alongside normal HTML tags. If you run the server now, you will see the following error page:

This is because we do not have any templates at all! Django looks for templates in a `templates` folder inside your app so go ahead and create one in your `howdy` app folder.

```
mkdir templates
```

Go into the templates folder you just created and create a file called `index.html`

```
(env) hello/helloapp/howdy/templates
 > touch index.html
```

Inside the `index.html` file, paste this code.

```
<!-- howdy/templates/index.html -->
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Howdy!</title>
    </head>
    <body>
        <h1>Howdy! I am Learning Django!</h1>
    </body>
</html>
```

Now run your server.

```
python manage.py runserver
```

You should see your template rendered.

Linking pages

Let's add another page. In your `howdy/templates` folder, add a file called `about.html`. Inside it, write this HTML code:

```
<!-- howdy/templates/about.html -->
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Howdy!</title>
    </head>
    <body>
        <h1>Welcome to the about page</h1>
    <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc quis neque ex. Donec feugiat egestas dictum. In eget erat
    </p>
    <a href="/">Go back home</a>
    </body>
</html>
```

Once done, edit the original `index.html` page to look like this.

```
<!-- howdy/templates/index.html -->
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Howdy!</title>
    </head>
    <body>
        <h1>Howdy! I am Learning Django!</h1>
     <a href="/about/">About Me</a>
    </body>
</html>
```

Clicking on the `About me` link won't work quite yet because our app doesn't have a `/about/` url defined. Let's edit the `urls.py` file in our `howdy` app to add it.

```
# howdy/urls.py
from django.conf.urls import url
from howdy import views

urlpatterns = [
    url(r'^$', views.HomePageView.as_view()),
    url(r'^about/$', views.AboutPageView.as_view()), # Add this /about/ route
]
```

Once we have added the route, we need to add a view to render the `about.html` template when we access the `/about/` url. Let's edit the `views.py` file in the `howdy` app.

```
# howdy/views.py
from django.shortcuts import render
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    def get(self, request, **kwargs):
        return render(request, 'index.html', context=None)

# Add this view
class AboutPageView(TemplateView):
    template_name = "about.html"
```

Notice that in the second view, I did not define a `get` method. This is just another way of using the `TemplateView` class. If you set the `template_name` attribute, a get request to that view will automatically use the defined template. Try changing the `HomePageView` to use the format used in `AboutPageView`.

Running the server now and accessing the home page should display our original template with the newly added link to the about page.

Clicking on the `About me` link should direct you to the `About` page.

On the `About me` page, clicking on the `Go back home` link should redirect you back to our original index page. Try editing both these templates to add more information about you.

Conclusion

That was just a brief look at how to get a Django website up and running. You can read more about Django on the official Django docs. The full code for this tutorial can also be found on Github.

Amos Omondi

2 posts

Full stack Python/Django software developer at Andela. Love discovering new things every day.