


Working with Python3.6 and dJango (Setup guide by RKtheDragon)

Working with Python 3.6.6 and dJango 2.0.7

Tested with Python 3.6.6 so I recommend to install this version rather than any other version.

	<p>Author: @RKtheDragon</p> <p>Working as Freelance Corporate Trainer for C, C++, Linux, SDLC, Symfony3, Laravel, PHP, MySql, MongoDB, Python3, Advance Java, AngularJS, NodeJS</p> <p>A Freelance WebMaster for Open Source Web Tools, having a total experience of more than 17 years in LAMP and WAMP.</p> <p>Email: rk07n@gmail.com</p>
---	--

Working with Django 2.0.7

While there are many possible commands we can use, in practice there are six used most frequently in Django development.

- `cd` (change down a directory)
- `cd ..` (change up a directory)
- `ls` (list files in your current directory, on *nix e.i. Linux, Unix)
- `dir` (list files in your current directory on DOS)
- `pwd` (print working directory, on Linux, Unix)
- `mkdir` (make directory)
- `touch` (create a new file, on *nix e.i. Linux, Unix)
- `copy con >> filename` (create a new file on DOS)
- `tree` (Tree structure of directory with folder and files, both in Dos and Linux)

For the beginners to Django, and Python, I must say that please have knowledge of being in path system, either you are working on which path. I specially say that here I am using two paths for installing and working.

- a) One path for installation of Python (version 3.6.6) so I named it Python36 in C: drive on Windows so path is **C:/Python36**
- b) The other path is **C:/wamp64/www/test/verEnv** which is my project directory. So keep focus on the path I have mentioned in this tutorial.

Follow the steps to work with Django 2.0.7

1. Installing Python, pip

- a. Install Python 3.6.6 (I have installed it in **C:\Python36** directory)
- b. Modify your path environment variable to include the location of the installed Python executable. To change environment variables in Windows 7:
 - i. Click the Windows start button in the lower-left corner of the screen.
 - ii. In the *Search programs and files* box, type *environment variables*
 - iii. When the search results appear, click *Edit the system environment variables*
 - iv. You should now see the *System Properties* window. Click *Environment Variables...*

Working with Python3.6 and dJango (Setup guide by RKtheDragon)

- v. When the *Environment Variables* window opens, choose *Path* from the *System variables* list and click *Edit...*
 - vi. Append the following location of the Python executable and the Python Scripts folder to the variable value, making sure everything is separated by a semicolon. For example, `;C:\Python36;C:\Python36\Scripts`
 - vii. Click *OK* after modifying the variable value, and click *OK* again to exit the *Environment Variables* dialog.
 - c. You should now be able to bring up an interactive Python shell by opening a command window and typing `python`.
2. Two such tools are `pip` and `easy_install` (part of `setuptools`). Despite some disadvantages on Windows, I chose `pip`. (as it comes with Python 3.6 installer)
3. Now we will setup virtual environment for running our dJango projects as below, before that go to `C:\Python36\Scripts` directory in the console to run this command:
 - a. **`pip install virtualenvwrapper-win`**
 - b. After successful installation of virtual environment give the following command to establish a new working space virtual environment, by going to your working directory:
 - i. `C:/wamp64/www/test/verEnv> mkvirtualenv my_django_env`
Here "`my_django_env`" is just a name, so you may name an environment any name like, `sona_env` here I have used `_env` as suffix, so this clears in your folder structure that this is your Environment Folder rather than any ordinary folder.
 - ii. Now you may see that you are in virtual environment as "`(my_django_env) C:/wamp64/www/test/verEnv>`"
 - iii. Now run the command to install dJango as below:

```
(my_django_env) C:/wamp64/www/test/verEnv> pip3 install django
```
 - iv. Check the version of dJango installed as:
`python -m django --version`
4. Now we will install mysql connection as below:
 - a. Install PyMySQL, by going in python executable directory and in scripts, in my case it is "`C:/Python36/Scripts`"
 - b. **`pip install protobuf`**
 - c. and then
 - d. **`pip install pymysql`** (This is specially for 32 bit computers with Python3.6 but we may try in 64 bit computers as well)

5. Now install **south** as below, by going in **C:/Python36/Scripts**:

a. `pip install south`

[South](#) is used for managing changes to your database tables. As your application grows, and you need to add a field to a specific table, for example, you can simply make changes to the database via migrations with South. It makes life *much* easier.

Now we don't need internet connection as all setup for **dJango** has been done.

Lets start work with dJango.

1. First exit from console and re-enter to it, to follow steps quietly:
2. Go to your working directory where you want to do scripting for your project
3. Now as we already created a virtual environment named "**my_django_env**" so, we will start working on this environment.
4. In your project directory, in my case its, "**C:/wamp64/www/test/verEnv/**" give the following command to work on virtual environment:

`workon my_django_env`

5. Now to create dJango project do as follow: (in **C:/wamp64/www/test/verEnv/**)

`django-admin startproject hello_world`

6. Here **hello_world** is the project name, which has created as new directory
7. Now give command "**tree**" which will show the file structure

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

1 directory, 5 files

Here the files and there working are as below:

The **settings.py** file controls our project's settings, **urls.py** tells Django which pages to build in response to a browser or url request, and **wsgi.py**, which stands for **web server gateway interface**, helps Django serve our eventual web pages. The last file **manage.py** is used to execute various Django commands such as running the local web server or creating a new app.

In the directory “C:/wamp64/www/test/verEnv/” we have a impression of “`pip freeze`” that all your libraries are installed, use the following command to create a record of the installed libraries within the “hello_world_project” directory:

```
pip freeze > requirements.txt
```

This will create a file named requirements.txt in “C:/wamp64/www/test/verEnv/” folder so it will show all the installations has been done so far in this project.

8. Now start the server

```
python manage.py runserver
```

This command will give you following message

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
You have 14 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
```

```
Run 'python manage.py migrate' to apply them.
```

```
December 29, 2017 - 03:03:47
```

```
Django version 2.0, using settings 'mytestsite.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

8. Once the server is running you can view the site by navigating to the following URL on your local web browser: <http://127.0.0.1:8000/>
9. If you want to change the server’s port, pass it as a command-line argument. For instance, this command starts the server on port 8080 as below:

```
python manage.py runserver 8080
```

Here 8080 is your port and <http://127.0.0.1:8080/> is web server address

If you want to change the server’s IP, pass it along with the port. For example, to listen on all available public IPs (which is useful if you are running Vagrant or want to show off your work on other computers on the network), use:

```
python manage.py runserver 0:8080
```

0 is a shortcut for **0.0.0.0**. Full docs for the development server can be found in the [runserver](#) reference

Automatic reloading of [runserver](#)

The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

10. Lets start with first page in view section:

```
python manage.py startapp pages
```

Here “**pages**” is your view directory, you may give a different name for views. The **tree** view of pages directory is as below:

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

Let's review what each new pages app file does:

- `admin.py` is a configuration file for the built-in Django Admin app
- `apps.py` is a configuration file for the app itself
- `migrations/` keeps track of any changes to our `models.py` file so our database and `models.py` stay in sync
- `models.py` is where we define our database models, which Django automatically translates into database tables
- `tests.py` is for our app-specific tests
- `views.py` is where we handle the request/response logic for our web app

Even though our new app exists within the Django project, Django doesn't “know” about it until we explicitly add it. In your text editor open the `settings.py` file and scroll down to `INSTALLED_APPS` where you'll see six built-in Django apps already there. Add our new pages app at the bottom:

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

```
    'pages', # new  
]
```

Views and URLConfs

In Django, *Views* determine **what** content is displayed on a given page while *URLConfs* determine **where** that content is going.

When a user requests a specific page, like the homepage, the URLConf uses a [regular expression](#) to map that request to the appropriate view function which then returns the correct data.

In other words, our *view* will output the text “Hello, World” while our *url* will ensure that when the user visits the homepage they are redirected to the correct view.

Let’s start by updating the `views.py` file in our `pages` app to look as follows:

```
# pages/views.py  
from django.http import HttpResponse  
  
def homePageView(request):  
    return HttpResponse('<p style="color:blue;">Hello, World!</p>')
```

Basically we’re saying whenever the view function `homePageView` is called, return the text “Hello, World!” More specifically, we’ve imported the built-in [HttpResponse](#) method so we can return a response object to the user. Our function `homePageView` accepts the `request` object and returns a response with the string `Hello, World!`.

Now we need to configure our urls. Within the `pages` app, create a new `urls.py` file.

Create a new file named `urls.py` in `pages` directory, and add the following code to it:

```
# pages/urls.py  
from django.urls import path  
  
from . import views  
  
urlpatterns = [  
    path('', views.homePageView, name='home')  
]
```

On the top line we import `path` from Django to power our `urlpatterns` and on the next line we import our `views`. The period used here `from . import views` means reference the current directory, which is our `pages` app containing both `views.py` and `urls.py`. Our `urlpatterns` has three parts:

- a Python regular expression for the empty string ''
- specify the view which is called `homePageView`
- add an optional url name of 'home'

In other words, if the user requests the homepage, represented by the empty string '' then use the view called `homePageView`.

We're *almost* done. The last step is to configure our project-level `urls.py` file too. Remember that it's common to have multiple apps within a single Django project, so they each need their own route.

Update the `helloworld_project/urls.py` file as follows:

```
# helloworld_project/urls.py
from django.contrib import admin
from django.urls import path, include

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

We've imported `include` on the second line next to `path` and then created a new urlpattern for our `pages` app. Now whenever a user visits the homepage at / they will first be routed to the `pages` app and then to the `homePageView` view.

It's often confusing to beginners that we don't need to import the `pages` app here, yet we refer to it in our urlpattern as `pages.urls`. The reason we do it this way is that that the method `django.urls.include()` expects us to pass in a module, or app, as the first argument. So without using `include` we *would* need to import our `pages` app, but since we do use `include` we don't have to at the project level!

Hello, world!

We have all the code we need now! To confirm everything works as expected, restart our Django server:

```
python manage.py runserver
```

If you refresh the browser for <http://127.0.0.1:8000/> it now displays the text "Hello, world!"

That's all you setup for working with dJango Project.