

Python virtual environment

In every programming language we expect our program is going on & on because it may have few reasons to say like this because our python packages is going to update frequently since their old functions are going to be useless and these methods are called as deprecated methods and the new powerful functions are comes in to the picture.

Now sometimes we want to freeze (use the fix versions of packages) the code and to this we need a venv.

Q: why we would we use virtual environment?

=> The purpose of VENV is the have a space where we can install packages that are specific to a certain project.

Ex: you have a lot of Django sites that use Django version 1. But now you have started use of Django version 2 for all of your newer projects. Now if you are just using your single global environment then when you update your Django version 2, it could break some of your old projects of Django version 1.

Now we don't want all our project pointing to the single instance of Django, so each project should have its own packages, separate from each other, so that's what we use VENV for.

So we create isolated environment for every project. So that it has only dependencies and package that they need and that's what the virtualenv allows to do.

Install virtual environment

```
>>pip install virtualenv
```

Q: How to use built-in venv module OR create VENV?

=>

- 1) Must required python 3.3 OR + version to create VENV.
- 2) Use 3.7 version

**** >> pip list**

This command will return list of all the global packages that you installed for your system & python.

```
C:\Users\asus\PycharmProjects\NewDjangoProject\Firstproject>pip list
```

Package	Version
absl-py	0.10.0
aiohttp	3.6.2
alabaster	0.7.12
altgraph	0.17
anaconda-client	1.7.2
anaconda-navigator	1.9.7
anaconda-project	0.8.3
appdirs	1.4.4
asgiref	3.2.10
asn1crypto	1.0.1
astroid	2.4.2
astropy	3.2.1
astunparse	1.6.3
async-timeout	3.0.1
atomicwrites	1.3.0
attrs	19.2.0
Automat	20.2.0
Babel	2.7.0
backcall	0.1.0
backports.functools-lru-cache	1.5

Create a directory in we create all the virtual environments.

>>mkdir Environments

Move into that directory

>>cd Environments

3) To create New VENV use below command.

>>python -m venv project_env

This will create a folder called your venv name 'project_env' with the sub directories: include, Lib, script & pyvenv.cfg.

(This will create a virtual mirror of python interpreter (cloned version of python interpreter), almost it creates a new python, so we need to give sanscar to this new born baby, so just install required packages for this baby and make it mature.)

>>virtualenv

=====

4) To check the directory of created venv use below command:

>> path>>dir

This will shows you your project_venv

```
C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>python -m venv project_env
```

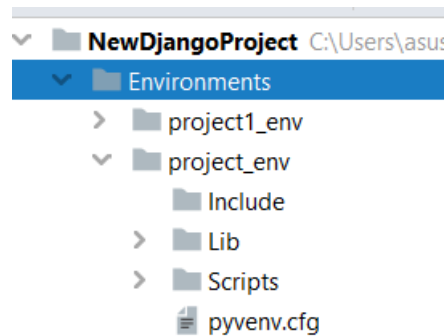
```
C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>dir
```

```
Volume in drive C is OS
```

```
Volume Serial Number is 8261-A571
```

```
Directory of C:\Users\asus\PycharmProjects\NewDjangoProject\Environments
```

```
05/24/2021  07:33 PM    <DIR>          .
05/24/2021  07:33 PM    <DIR>          ..
05/24/2021  07:31 PM    <DIR>          project1_env
05/24/2021  07:33 PM    <DIR>          project_env
               0 File(s)                0 bytes
               4 Dir(s)  135,927,246,848 bytes free
```



5) To activate that VENV use below command:

```
>>project_env\scripts\activate.bat
```

Now if this command will show you that is This project uses project_env
OR Not like:

```
>>(project_env) C:\users\asus\pycharmprojects\Durga>
```

it means right now you are using the 'project_env' for your project.

```
C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>project_env\scripts\activate.bat
```

```
(project_env) C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>
```

6) To check the directory of our virtual environment use below
command:

```
>>where python
```

```
C:\Users\asus\PycharmProjects\Durga\project_env\Scripts\python.exe
```

```
C:\Users\asus\Anaconda3\python.exe
```

```
C:\Users\asus\AppData\Local\Programs\Python\Python37\python.exe
```

Try these:

```
>>which python
```

```
>>which pip
```

7) Now to check the all the available libs in new venv use below
command:

```
>> pip list
```

Package	Version
---------	---------

pip	19.0.3
setuptools	40.8.0

```
(project_env) C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>pip list
```

Package	Version
pip	19.0.3
setuptools	40.8.0

It won't contains all the global packages.

8) From now if you install any packages here, then they will only be installed in this new environment only.

Ex: your project required request & pytz

>> pip install request

>> pip install pytz

Check most frequently used modules in python with their usage for interview.

9) After this installation again we use the same command to check libs of same venv.

>> pip list

it will show all the packages within our new venv

10) Now if you ever want to export those packages that you used for a specific project environment, we use **requirement.txt** file which this will allow someone else to create same environment used for this project

and use that requirement.txt, to install all of these same packages and dependencies that you are using. Use below command.

>> pip freeze --local >requirements.txt

this is similar to >>pip list. But this will give us a packages in the correct form for requirements.txt file. So that you can take that output & put that output into requirements.txt file. (This will create requirements.txt file in your directory and if you already have a requirements.txt then just use the freeze command.)

Note:→ Now install few packages in this environment and then make freeze and check installed packages.

11) Create requirement.txt file in our directory & put info from

'>>pip freeze' inside of there.

Do it manually. Create requirement.txt file & pass all the libs got from '>>pip freeze' command.

12) How to deactivate our Venv.

>>deactivate

by this command our below command:

(project_env) C:\Users\asus\PycharmProjects\Durga>

get converted into:

C:\Users\asus\PycharmProjects\Durga>

```
(project_env) C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>deactivate
C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>pip freeze
```

13) if you want to delete venv all together, so it's no longer active but it still, just delete directory for that environment use command:

>>rmdir project_env /s

'/s' → will delete entire tree
it will ask for confirmation, press **Y**
Now this environment should be deleted.

```
C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>rmdir project1_env /s  
project1_env, Are you sure (Y/N)? Y
```

Steps:→

Step-1:

* Generally venv is created inside our project & we named that as 'venv'
it's a very common convention. Because there is a difference between
venv module & name of the environment.

Ex:

```
>>mkdir my_project
```

Step-2:

Create venv inside of this project folder called as venv

```
>> python -m venv my_project\venv
```

It might take a seconds to create this venv.

Step-3:

Activate that venv

```
>>my_project\venv\scripts\activate.bat
```

Step-4:

if your project required the same modules OR libs that is used for above
project then we already used the requirement.txt file of last project
which has list of all those packages. Just use that

```
>>pip install -r requirement.txt
```

this will install everything from that requirement.txt file & all of the
dependencies that we has listed.

Step-5:

Once all these are installed, check all list using:

```
>>pip list
```

OR

```
>>dir
```

* Its general to keep the venv in your project directory.

Enter into your project

```
>>cd my_project
```

Now whatever the project modules OR files OR directories that you create, we would never put them inside venv directory. Create a simple script (py file) & put it into root of this directory (outside of venv).

* Should not commit your venv to source control. Because of this it will ignore the venv. So what would you commit the source is files like: **requirements.txt** file which let other people will built their own environment to run your project. But there is no need to add the environment itself.

Now deactivate environment:

```
>>deactivate
```

```
>>rmdir venv /s →delete previous directory with subdirectory Y
```

* Create venv by using system package access:

```
>> pip list
```

List of all system packages of python to install globally

```
>>python -m venv venv --system-site-packages
```

First venv is module name.

Second venv is our venv

--system-site-packages →for this will create environment with the system packages. so activate that

```
>> venv\scripts\activate.bat
```



```
>>pip list
```

See list of system packages are available inside our venv,
and also some addition packages are won't effect on system package
>>dir

But in above environment we dont have sqlalchemy in global installation
of python, so we can install it now.

```
>>pip install SQLAlchemy
```

This will also install in our enviro nment

```
>> pip list
```

now SQLAlchemy is also available here.

But it is also possible to see only those packages that are installed in this
venv use:

```
>>pip list --local
```

show only our venv packages.

```
(venv) C:\Users\asus\PycharmProjects\NewDjangoProject\Environments>pip list --local
Package      Version
-----
pip           19.0.3
setuptools   40.8.0
```

But if we deactivate our venv then it will remove the SQLAlchemy from
system packages also.

```
>>deactivate
```

now check system packages, where SQLAlchemy is not present

```
>>pip list
```

=====

Also install multiple packages at once using pip:

```
>>pip install pytz numpy sqlalchemy etc.
```

When you are working with the multiple project at once then its easy to get your packages and dependencies mixed up and also its hard to keep track of venv and venv variables overriding each other. So lets see how to keep all of these separate and easy to work with.

Conda is similar to pip in anaconda.

pipenv

Check current version of django and flask with other packages too.

How to use pipenv?

It's a new way to combine package management with venv. And is a highly recommended tool by python.org. This will combine a feature of pip and venv.

Pip → is used to add/install additional packages to project to provide lot of functionality to python.

Venv → a specific virtual environment for each project.

But pipenv → will simplifies the process of above.

Step:1 → install pipenv

```
>>pip install pipenv
```

Step:2 → start new project and navigate to that location

```
>>cd my_project/
```

Now at this point if we are using the older method to create **venv** then we need to activate the environment manually and after need to install all the packages. But with **pipenv** this is build in together now.

****** So instead of creating new environment manually, we simply just install the packages with **pipenv**.

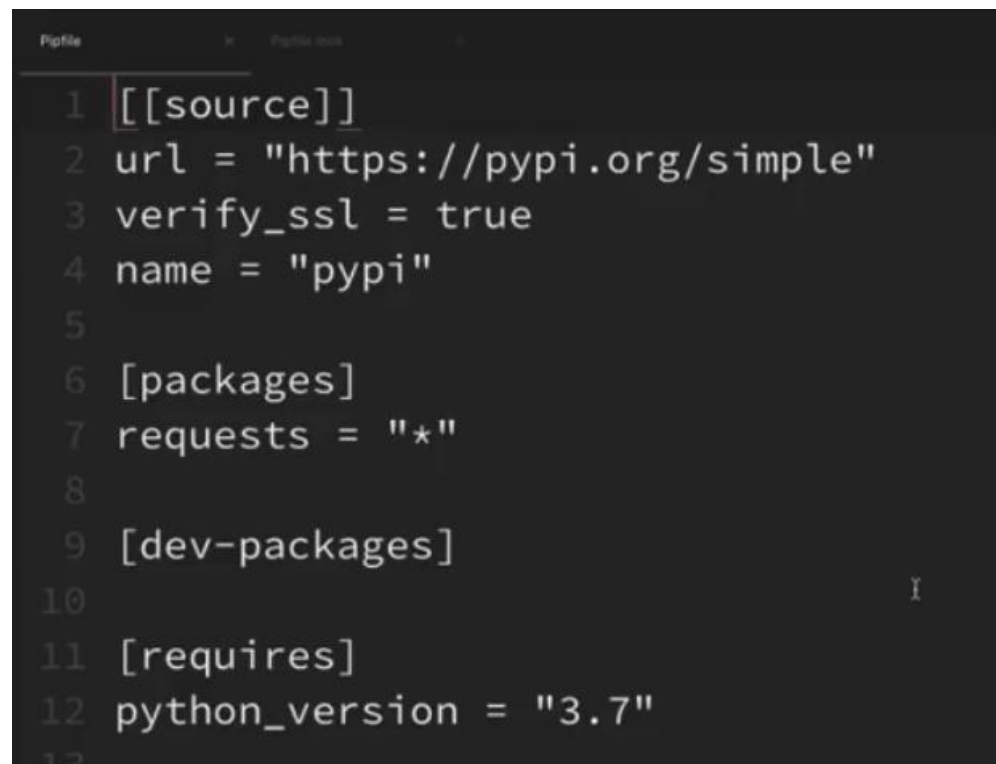
Step:3→install packages using **pipenv**

>>**pipenv install requests**

This will create **venv** for this project automatically, also provide a location of **pip** file (**pip** file describe our environment, this is like a **reuirements.txt** file & is a replacement of **requirements.txt** file) by using which version of python. This will also provide a location of **venv**.

This will also create a **pipfile.lock**, so don't touch it.

Look for pipfile:

A screenshot of a code editor showing a Pipfile configuration. The file is titled 'Pipfile' and contains the following content:

```
1 [[source]]
2 url = "https://pypi.org/simple"
3 verify_ssl = true
4 name = "pypi"
5
6 [packages]
7 requests = "*"
8
9 [dev-packages]
10
11 [requires]
12 python_version = "3.7"
```

url→ it tells us from where we are downloading the packages from.

Packages→

Requests==*" → it tells that we didn't specified the version while installing packages. (But if you specify then it will auto update the version, but the auto updatation of packages because we don't wat updates will break our project since this is risky and since the **pipfile.lock** is there.)

Requires→ it tells us which version of python we are using.

Pipfile.lock→

```
1 {
2   "_meta": {
3     "hash": {
4       "sha256": "bb57e0d7853b45999e47c163c46b95bc2fde31c527d8d7b5b5539dc979444a6d"
5     },
6     "pipfile-spec": 6,
7     "requires": {
8       "python_version": "3.7"
9     },
10    "sources": [
11      {
12        "name": "pypi",
13        "url": "https://pypi.org/simple",
14        "verify_ssl": true
15      }
16    ]
17  },
18  "default": {
19    "certifi": {
20      "hashes": [
21        "sha256:339dc09518b07e2fa7eda5450740925974815557727d6bd35d319c1524a04a4c",
22        "sha256:6d58c986d22b038c8c0df30d639f23a3e6d172a05c3583e766f4c0b785c0986a"
```

This is quit complex, which is generated file so we cant do changes manually. It contains a exact version if packages that we installed.

```

37     ],
38     "version": "==2.7"
39 },
40 "requests": {
41     "hashes": [
42         "sha256:63b52e3c866428a224f97cab011de738c36aec0185aa91cfacd418b5d58911d1",
43         "sha256:ec22d826a36ed72a7358ff3fe56cbd4ba69dd7a6718ffd450ff0e9df7a47ce6a",
44     ],
45     "index": "pypi",
46     "version": "==2.19.1"
47 },
48 "urllib3": {
49     "hashes": [
50         "sha256:a68ac5e15e76e7e5dd2b8f94007233e01effe3e50e8daddf69acfd81cb686baf",
51         "sha256:b5725a0bd4ba422ab0e66e89e030c806576753ea3ee08554382c14e685d117b5",
52     ],
53     "version": "==1.23"
54 }

```

Like request package has exact version. So the benefit of this file is this will always gives you exact environment everytime. And if any updates are available for any package then it will update it accordingly. So after testing your projects with updated packages then make the changes in lock file also therefore in future it will gives you those packages with updated version only and once the lock file ready with updated packages then you can push it on production.

Step:4 → activate the pipenv

>>pipenv shell

Step:5 → check python in this environment

>>python

Step:6 → check the venv is created in put executable file or not.

>>import sys

```
>> sys.executable
```

Means whatever we install in this folder that will separate from our other projects from this machine.

Step:7 → to come out of it use

```
>>exit()
```

Step:8 → deactivate the venv

```
>>exit
```

Here if we use the deactivate command then it looks like it does something but it won't actually exit completely, because pipenv actually launches subshell to activate the env and to exit that completely we need to use the exit command.

Step:9 → now we can run commands without activating venv

```
>>pipenv run python
```

Step:10 → we can also run the python script using pipenv like:

```
>>pipenv run python script.py
```

Step:11 → now we can install multiple packages at a time using requirements.txt (list of dependencies and packages required for project) file. (as already know how to install one at a time).

But make this requirements.txt file available in your project directory

```
>>pipenv install -r requirements.txt
```

```
>> pipenv install -r ../path/requirements.txt
```


Step:13→ how to freeze the file using pipenv?==> here we use the lock file

>>pipenv lock -r

```
$ pipenv lock -r
-i https://pypi.org/simple
certifi==2018.10.15
chardet==3.0.4
django-crispy-forms==1.7.2
django==2.1
idna==2.7
pillow==5.2.0
pytz==2018.5
requests==2.19.1
urllib3==1.23
```

This will show the installed dependencies for projects with their versions.

Step:13→ now we can also install some packages that are needed in development environment but not in production.

>> pipenv install pytest --dev

Step:14→ now open the pipfile and check the dev-packages category it will show the pytest package with version.


```
Pipfile
1 [[source]]
2 url = "https://pypi.org/simple"
3 verify_ssl = true
4 name = "pypi"
5
6 [packages]
7 requests = "*"
8 django-crispy-forms = "==1.7.2"
9 pytz = "==2018.5"
10 Django = "==2.1"
11 Pillow = "==5.2.0"
12
13 [dev-packages]
14 pytest = "*"
15
16 [requires]
17 python_version = "3.7"
18
```

But is not available in [packages] category because [packages] contains those packages which are required in production.

Step:14→ we can easily remove the installed package if you want.

>> pipenv uninstall pillow

This will uninstall and remove from [packages]

Step:14→ if we want to change the version of python for this environment.

Way:1→ you can directly make changes in pifile and recreate a venv using command:

```
>>pipenv -python 3.8
```

This will remove the existing venv and create this new venv and also uses python 3.6, check by:

```
>> pipenv run python
```

Step:14→ we can remove the venv completely

```
>>pipenv - -rm
```

This will only remove the venv that we are using.

Step:15→ Now we have to recreate that venv that matches to our pipfile:

```
>> pipenv install
```

This will create environment from scratch using python version and packages from pipfile.

Step:16→get the path of venv

```
>>pipenv - -venv
```

Step:17→ we can check the know security vulnerability for our installed packages.

```
>>pipenv check
```

```

$ pipenv check
Checking PEP 508 requirements...
Passed!
Checking installed package safety...
36522: django <2.1.2,>=2.1 resolved (2.1 installed)!
An issue was discovered in Django 2.1 before 2.1.2, in which unprivileged users can
read the password hashes of arbitrary accounts. The read-only password widget used
by the Django Admin to display an obfuscated password hash was bypassed if a user
has only the "view" permission (new in Django 2.1), resulting in display of the ent
ire password hash to those users. This may result in a vulnerability for sites with
legacy user accounts using insecure hashes.

36517: django <2.1.2,>=2.1.0 resolved (2.1 installed)!
django before 2.1.2 fixes a security bug in 2.1.x.
If an admin user has the change permission to the user model, only part of the
password hash is displayed in the change form. Admin users with the view (but
not change) permission to the user model were displayed the entire hash.

```

This will also shows some issues of django with this version which may solved in upcoming version.

Step:18→whenever you make any changes in pipfile like update any version of package manually then must run the command like:

>> pipenv install

Which will reinstall all the packages from scratch which are available in pipfile. This will also update pipfile.lock

Now try this:

```

$ pipenv check
Checking PEP 508 requirements...
Passed!
Checking installed package safety...
All good!

```

Step:19→pipenv also provide a dependency graph which tells us packages and their dependencies.

>>Pipenv graph

This will shows required dependent packages for existing packages.

Step:20→what to do while pushing this code on production:

→Out pipfile.lock contains all the exact hashes and versions for specific packages that we currently using in this project. So before production we have to test our code with the existing dependencies OR check whether we need an updated version of packages and once its confirm and tested successfully with required package versions then make likewise changes in **pipfile.lock** too by saying:

>>pipenv lock

```
$ pipenv lock
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (ec8fd2)!
```

Now you can take your pipfile.lock and move it into production environment.

Now simply run the command:

>>pipenv install --ignore-pipfile

This will ignore the pipfile packages and create environment with the packages available in pifile.lock.

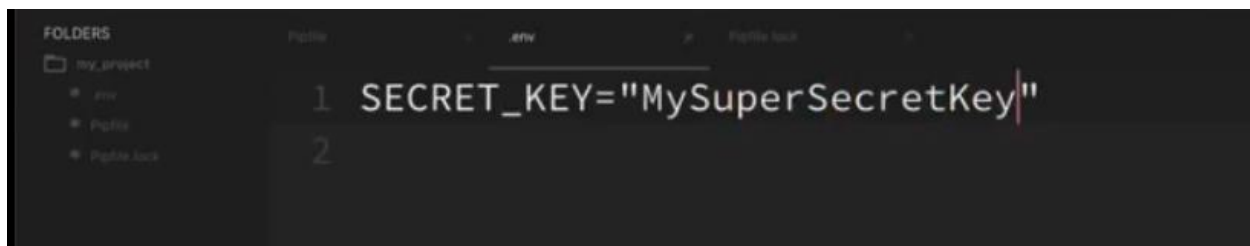
Set environment variable to specific venv. This is imp when you are working with the multiple Flask and Django projects and want to have en variable for each project that contains project secrete key, DB connection info thinks like that then it will be nice to have venv to handle this without needing to do that manually.

So to do that using pipenv we need to create .env file in put project directory, and we can set the env variable in that file OR multiple environment variables in that file.

Create new file with name .env

And here we add the environment variable that will accessible in our environment.

Like add secrete key in that file:



Now go to terminal and check the env var that we created recently:

```
>>pipenv run python
```

To get env variable use os module:

```
>>import os
```

```
>>os.environ['SECRETE_KEY'] #use name of key
```

```
$ pipenv run python
Loading .env environment variables...
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.environ['SECRET_KEY']
'MySuperSecretKey'
```

Note:→ Don't commit that pipfile OR .env file to your repository like on Git.

Because it may have a sensitive info.

** If you want to test the installed packages by importing, then very first need to open python by command:

```
>>python
```

Now you can try module/package importing.

Requirements.txt: ➔ This file contains required packages for your project, and if you want to share this project to someone else then instead of installing every package at once/individually we just install all the packages at once using requirements.txt file. So it will become easy to replicate whole project.

Here two commands are imp:

- 1) To save all packages in requirements.txt with versions.

```
>> pip freeze --local > requirements.txt
```

OR

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

- 2) Install all the packages present in requirements.txt file for same project but on other system/person.

```
>> pip install -r .\requirements.txt
```

- 3) To bring all the system packages in your venv then use below command:

First deactivate the current venv and then use the below command:

```
>> virtualenv --system-site-packages venv_2
```

This will create a directory named as **venv_2** which contains all the system packages.

- 4) Now activate venv_2

```
>> .\venv_2\scripts\activate.bat
```

- 5) Now if we import flask then it will bring this flask from system interpreter packages

```
>> python
```

```
>> import flask
```

- 6) >>exit() ➔ used to come out of python interpreter

7) >> deactivate → used to come out of venv
