Python Packages - installing and using.

Barry Rowlingson

Lancaster University Medical School

CHICAS centre for health informatics, computing, and statistics

A big part of Python is working with other people. This talk is all about how we can use other people's code!

PyPi

Python Package Index

https://pypi.org/

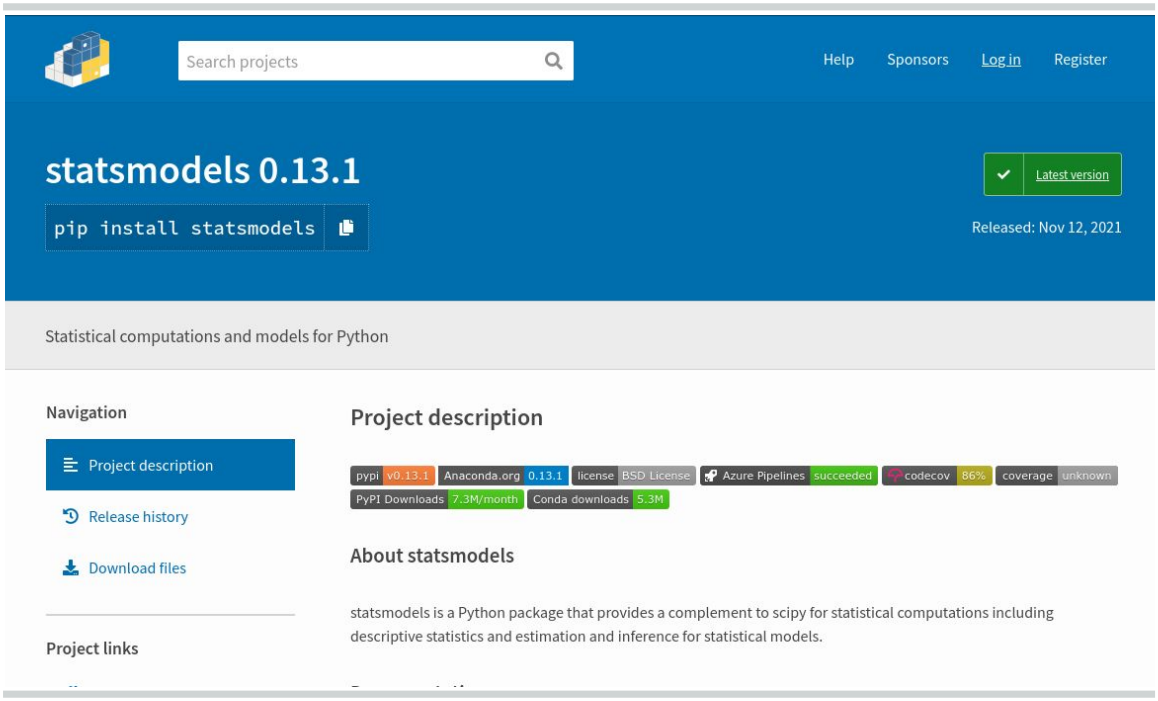
The main source of good stuff to add to your basic python installation can be found in the Python Package Index, or PyPI.

PyPi

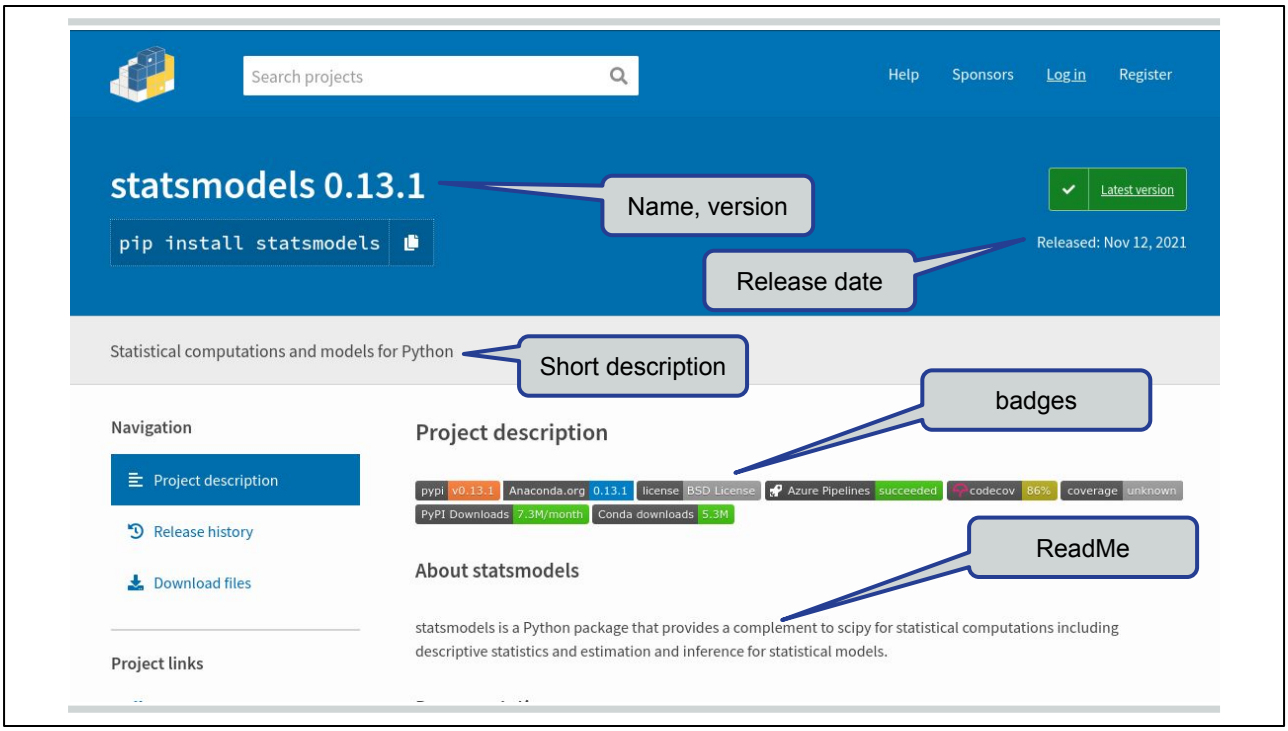
CHICAS centre for health informatics, computing, and statistics

- Lists 340,000 *projects*
- Index and metadata only - doesn't store the code
- Code will be on github, GitLab, etc.
- No quality control
- No security checks

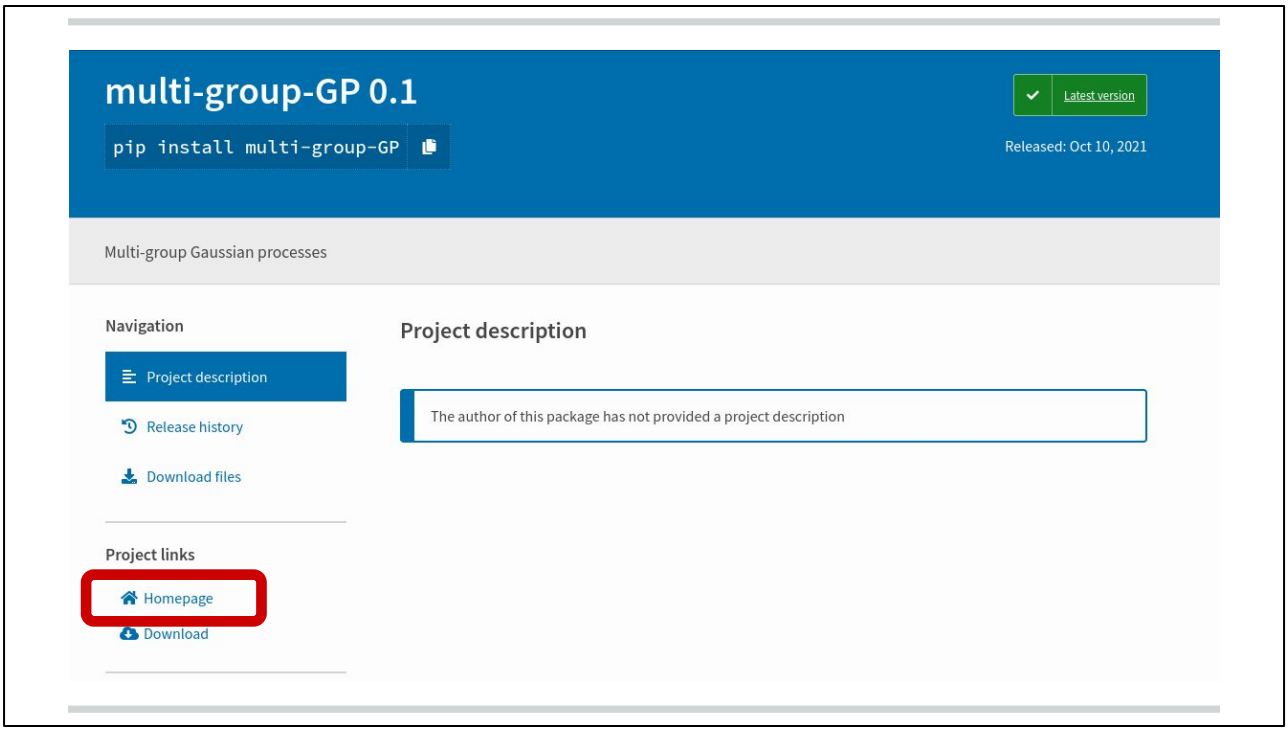
PyPI is an index of python packages, and it is managed by the Python Software Foundation (PSF). This body is independent of the language developers. There's over 300,000 packages listed, but note it is only an index, the actual source code for those packages are to be found elsewhere, usually a coding site like github or GitLab. Also note there's no quality control or security checks so there's few guarantees that packages will actually work...



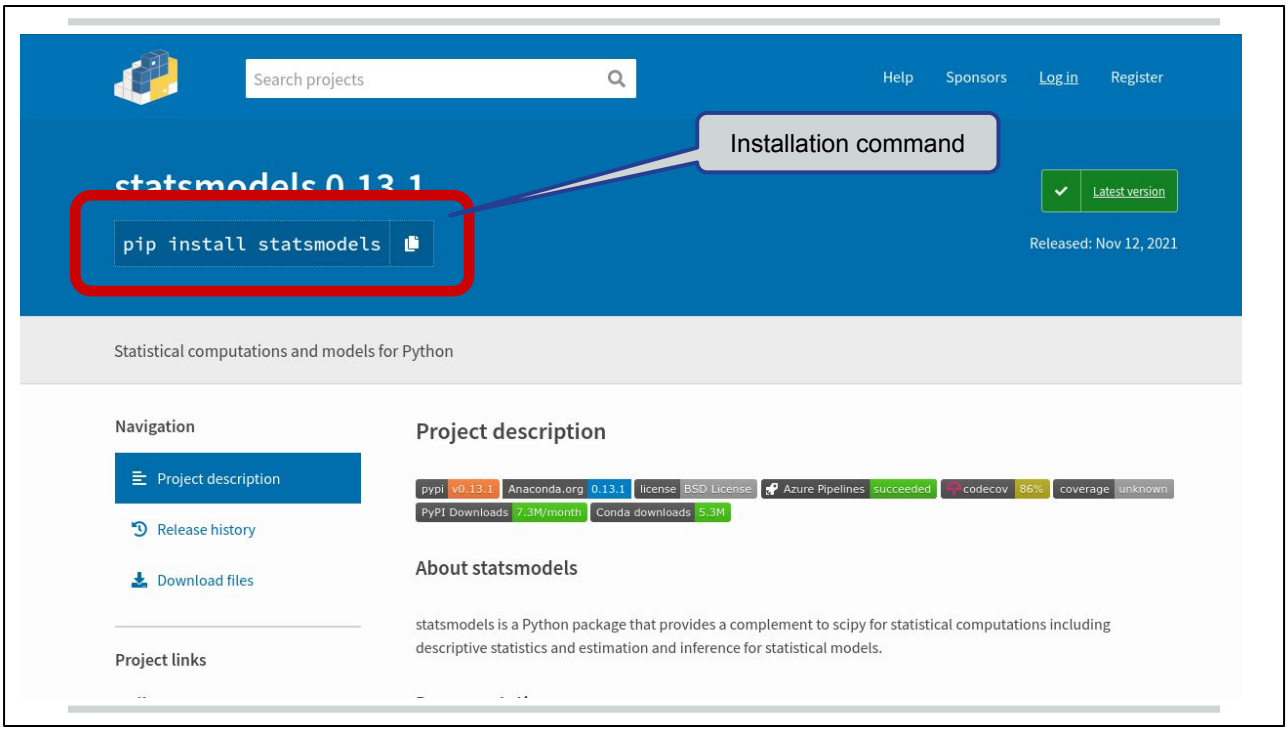
A typical project page might look like this. This is statsmodels index page on pypi.org - you can see lots of things here...



At the top, the name, the version, and the release date for this version. Then a one-line description and a bunch of “badges” showing all sorts of info such as the version, the license type, various test statuses, download counts etc. These badges will often appear in markdown files for the package. Then there’s the readme and a sidebar of links.



statsmodels is a very complete index page, this is a minimal one with just a name, version, release data and short description. All the substance of this package will be found in that homepage link.



One thing the page gives you is the command to install the package in your system. Right here is the “pip” command. You can even click on the copy icon and then paste it into your system.

pip

- Python package installer

```
C:> pip

Usage:
  pip <command> [options]

Commands:
  install                     Install p
  download                    Download
  uninstall                   Uninstall
  freeze                      Output ir
  list                        List inst
```

```
C:> python -m pip

Usage:
  C:\Users\oswal\AppData\Local\Programs\

Commands:
  install                     Install pa
  download                    Download p
  uninstall                   Uninstall
  freeze                      Output ins
```

Pip is the package installer. You can either run it as a command on your system command line or run it from python using the -m option and the module name.

Dependencies

All packages have a file that lists the other packages they need:

```
[build-system]
requires = [
    "setuptools",
    "wheel",
    "cython>=0.29.22",
    "numpy==1.17.4; python_version=='3.7'",
    "numpy==1.17.4; python_version=='3.8'",
    "numpy==1.19.4; python_version=='3.9'",
    "numpy; python_version>'3.9'",
    "scipy>=1.3",
]
```

Most packages need other packages to make them work, so they include a file of their dependencies, including versions and options.

Install

Installation checks and gets dependencies:

```
C:> pip install statsmodels
Collecting statsmodels
  Downloading statsmodels-0.13.1-cp310-none-win_amd64.whl (9.5 MB)
     |                                | 9.5 MB 125 kB/s
Requirement already satisfied: patsy>=0.5.2 in c:\users\oswal\appda
site-packages (from statsmodels) (0.5.2)
Requirement already satisfied: scipy>=1.3 in c:\users\oswal\appdata
te-packages (from statsmodels) (1.7.2)
Requirement already satisfied: numpy>=1.17 in c:\users\oswal\appdat
ite-packages (from statsmodels) (1.21.4)
```

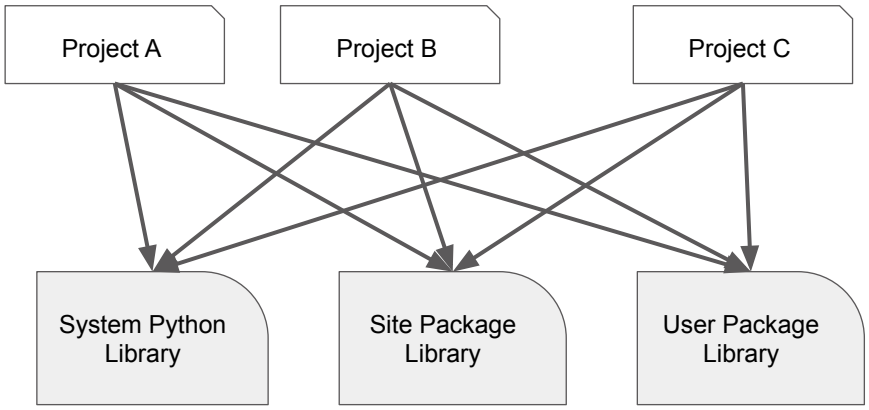
This means that when you install a package it may also get a bunch of other packages. Here I'm installing statsmodels and all these dependencies are already on my system so it doesn't get them.

What's Installed?

- Python modules
  - Things to import from other python code
  - Help text etc
- New commands
  - Things to run at the command line - like the `pip` command itself.

When you install a python package you get a few things: first the code will probably be a module that you can import into your own code, together with its help documentation. It can also create new system commands. The pip command itself is a command that comes from the installation of the "pip" module, which comes with python (otherwise you'd have to install it, and you wouldn't have pip to install pip..)

Libraries

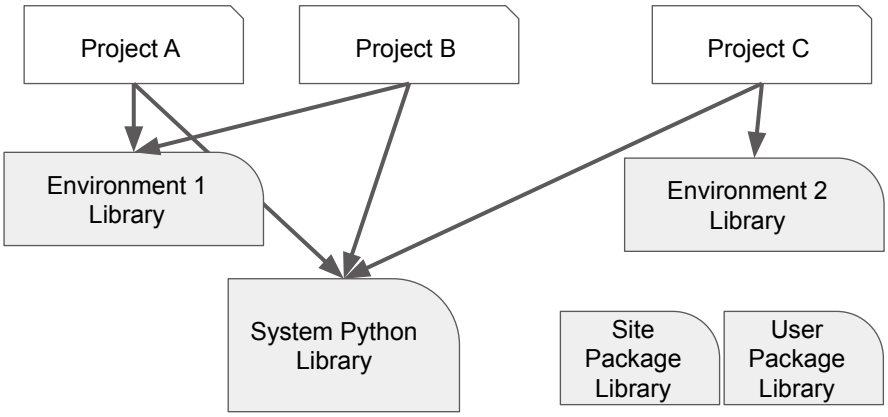


Python modules get installed into libraries, which are folders on your computer containing the code. There's at least three libraries for a python installation - the system python library which contains the fundamental modules that were installed when python was installed on your system. Then if the administrator or system user has installed extra modules they'll be found in the "site packages" library. These are available to all users of the system. Then if a user installs a module themselves (by running pip with no special privileges) they get installed into the user's personal package library. Any python code will look in these three locations when it runs "import something".

The downside to this is apparent if any projects you might be working on need a different version of a package to another project. For example you might need the very latest version of a package, or even an unreleased development version. But you don't want to update it and possibly all its dependencies for everything else that you are doing.

So an extra level of library structure has been developed based around "virtual environments".

Virtual Environments



When a virtual environment is in play, packages are installed and imported from a new folder which contains the library for the environment. The site and user package library are not used, and so you are starting from a clean system environment. Here projects A and B are sharing packages from “Environment 1” and project C is using a separate environment “Environment 2”. Nothing that projects A and B do can affect the packages that project C is using, and vice versa.

```
PS C:\Users\oswal\Work\Venvs> dir
PS C:\Users\oswal\Work\Venvs> python -m venv e1
PS C:\Users\oswal\Work\Venvs> dir e1


    Directory: C:\Users\oswal\Work\Venvs\e1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        24/11/2021     16:19                Include
d-----        24/11/2021     16:19                Lib
d-----        24/11/2021     16:19                Scripts
-a----        24/11/2021     16:19            119 pyvenv.cfg


PS C:\Users\oswal\Work\Venvs> .\e1\Scripts\activate
(e1) PS C:\Users\oswal\Work\Venvs>
```

Starting from empty folder

Create new venv **e1**...

..is a folder with things in

Run the activate script

So how do we make and use virtual environments? There's a python module for that. Running `python -m venv <name>` creates a new virtual environment with that name in the current working folder. You'll see a config file and some folders with a few things in.

To start using the environment, you run the "activate" script in the folder. A subtle change happens...

```
PS C:\Users\oswal\Work\Venvs> dir
PS C:\Users\oswal\Work\Venvs> python -m venv e1
PS C:\Users\oswal\Work\Venvs> dir e1


    Directory: C:\Users\oswal\Work\Venvs\e1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        24/11/2021     16:19                Include
d-----                       16:19                Lib
d-----                       16:19                Scripts
-a----          /11/2021     16:19            119 pyvenv.cfg


PS C:\Users\oswal\Work\Venvs> .\e1\Scripts\activate
(e1) PS C:\Users\oswal\Work\Venvs>
```

Change of prompt shows the venv we are in!

Here the prompt now starts with the name of the environment in green at the start. This shows we are now in a virtual environment called “e1”.

```
(e1) PS C:\Users\oswal\Work\Venvs> python
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [M
MD64)] on win32
Type "help", "copyright", "credits" or "license" for more info
>>> import statsmodels
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'statsmodels'
>>>
```

Uh oh! Where's **`statsmodels`** gone?

Needs installing into this virtual environment

```
(e1) PS C:\Users\oswal\Work\Venvs> pip install statsmodels
Collecting statsmodels
  Using cached statsmodels-0.13.1-cp310-none-win_amd64.whl (9.5 MB)
Collecting scipy>=1.3
  Downloading scipy-1.7.3-cp310-cp310-win_amd64.whl (34.3 MB)
     |                                | 12.1 MB 731 kB/s eta 0:00:31
```

So now I run python and import statsmodels. Which fails, because statsmodels was in my user library and that isn't searched in a virtual environment. So I have to install it again, and pip installs it into the environment.

Now I see it

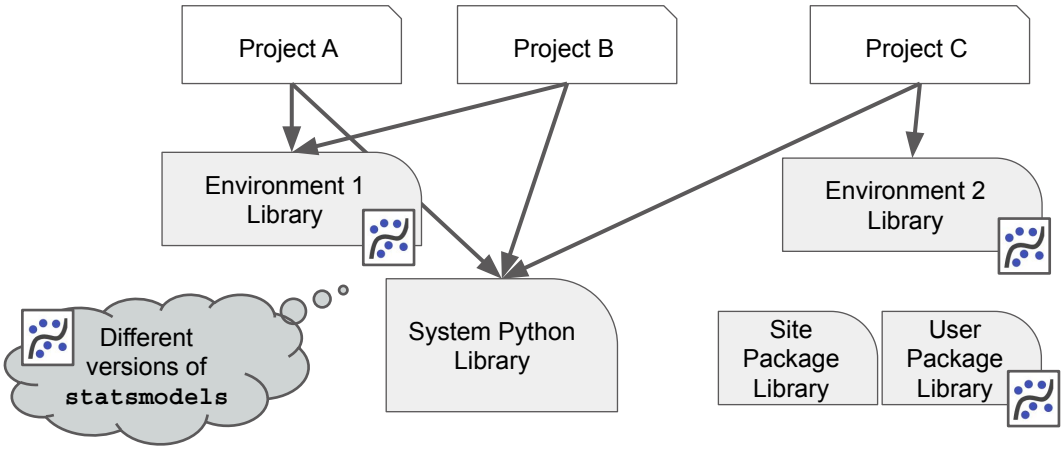
```
(e1) PS C:\Users\oswal\Work\Venvs> python
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (A
MD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import statsmodels
>>> statsmodels.__file__
'C:\\Users\\oswal\\Work\\Venvs\\e1\\lib\\site-packages\\statsmodels\\__init__.py'

>>> _
```

The **`__file__`** element shows me which python file this has come from.

Now I can import it, and if I look at the __file__ element I can see where the file is, and its below the `e1` folder that was created earlier, ie it is in the virtual environment library.

Virtual Environments



What I've got now is something like this. There's a statsmodels installation in my user library, and other installations in environment libraries. When a virtual environment is active, that's the one that is seen. In theory I could have different versions of statsmodels all over the place.

Freeze/Restore

```
(e1) PS C:\Users\oswal\Work\Venvs> pip freeze
numpy==1.21.4
pandas==1.3.4
patsy==0.5.2
python-dateutil==2.8.2
pytz==2021.3
scipy==1.7.3
six==1.16.0
statsmodels==0.13.1
(e1) PS C:\Users\oswal\Work\Venvs>
```

List site packages and versions with **`pip freeze`**

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

Restore those packages at those versions with **`pip install -r reqs.txt`**

These venvs are a great aid to reproducibility, since there's a "pip freeze" command which lists all the packages and their version numbers. If you send this to a file, and give the file to someone, they can create a venv with all those packages, and those versions, themselves, and run your code in an identical environment to you. Which means your code should work!

Finished?

Deactivate:

```
(e1) PS C:\Users\oswal\Work\Venvs> deactivate
PS C:\Users\oswal\Work\Venvs>
```

No `e1` any more - back to the system site package library

Once you're done with working in an environment you "deactivate". Its still there with all the installed packages in it, waiting for you to activate it again. You'll see the prompt has reverted back to plain and python will now search in the system site and user package libraries.

Conda/Anaconda

- System for
  - Creating venvs
  - Installing packages into venvs
  - Reproducing venvs with packages

Some python installs have their own virtual environment functions and installers, if you are using conda or anaconda I think there's command for all this from the "conda" command...

Conclusion

- PyPI is the Python package index
- `pip` installs packages[1]
- Use virtual environments

1. pip stands for "pip installs packages"

Three Main Bullet Points: PyPI is the package index ; pip installs packages ; understand virtual environments for package installation.