Lecture #7: Modules and Venv

Presented by Jamal Bouajjaj

2023–10–02

For University of New Haven's Fall 2023 CSCIxx51 Course

Modules

Expanding Python

While the built-in Python functions and what you can do might be fine for some applications, what if you want to expand the functionality of you application?

For example, what if you need to get the factorial of a number? Sure, you can make a function, but does it already exist?

Modules

The answer is with Modules. Modules are python files or packages than can be imported into your application.

A Package is just a collection of modules. All packages are modules, not all modules are packages.

Importing

Say you have a two python files: `main.py` and `config.py`. From main, you can import config with the import statement:

```
import config
```

`config` is now imported as an object containing all functions, classes, and variables from `config.py`

name

If `config.py` had some things executed (i.e code that runs if the file is ran, isn't a declaration), importing the file will cause that code to run, which is not intentional.

name

If `config.py` had some things executed (i.e code that runs if the file is ran, isn't a declaration), importing the file will cause that code to run, which is not intentional.

What if you need to be able to run the script independently, but also able to import it?

name

If `config.py` had some things executed (i.e code that runs if the file is ran, isn't a declaration), importing the file will cause that code to run, which is not intentional.

What if you need to be able to run the script independently, but also able to import it?

This is where `__name__` comes in handy:

```
if __name__ == "__main__":
  DO SOMETHING
```

Import statement

The full syntax for import is `import MODULE as NAME` or `from MODULE import IDENTIFIER as NAME`.

`from` is used if you want to import something from a module, for example `from random import randint`.

The usage of as is optional, and simply renames what you import, for example `import numpy as np`.

Import ALL

From a module, you can import so that every namespace is directly available in your application, for example

```python
from math import *
print(pi)   # Is now in our local namespace
```

This is generally not the best idea for multiple reasons: Namespace collision and inefficient for big packages.

Generally, it is better to explicitly import what you need, in the case above

```python
from math import pi
print(pi)   # Is now in our local namespace..again
```

Import search

The import function searches the following places for modules:

- The current directory
- sys.path

sys.path is initialized automatically by Python, and has the following directories added (as over simplification):

- Everything in the PYTHONPATH environment variable, if it exists
- Standard modules that comes with Python
- Site packages

Standard Modules

Python has a LOT of standard modules that it comes with.

They can be found in
`https://docs.python.org/3/library/index.html`

Site Packages

You can download external modules and install them into your site packages directory.

This process is done with a built-in tool called `pip`.

PIP

pip is a tool to allow to install site packages. You can either download the "compressed" package (in a wheel format) and install it with

```
pip install DOWNLOADED.whl
```

Or you can have pip automatically download the module for you! The available modules can be found at `https://pypi.org/`. The download syntax is the same, for example

```
pip install numpy
```

venv

Dependency Hell

What if for one project you need `pymongo==4.5.0` for for another project you need `pymongo==3.13.0`?

You can't have your system site package include both: they conflict with another!

No!

What if you can have a local environment that includes all dependencies required by your project, but nothing else?

Like a sandbox of dependencies

Virtualize the Environment

This does exist: Virtual Environments.

As site packages is just a path where to live, virtual environments can point the site-package directory to a local folder so it doesn't interfere with system packages.

A virtual environment also has an internal sym-link to the correct system python interpreter version, for example 3.7.10 vs 3.11.0, so your virtual environment also can dictate the python version it wants to run as (must still be installed system-wide).

Make it!

To make one, call the Python `venv` module and specify the path to make the virtual environment in:

```
python -m venv .venv
```

Use it!

To use it, activate it by running ‘activate.bat‘ if using CMD

```
.venv/Script/activate.bat
```

or the .ps1 file if using powershell

```
.venv/Script/activate.ps1
```

To source the activate file if using a reasonable shell (Linux or MacOS):

```
source .venv/bin/activate
```

Bop it!

Now when you call `python` or `pip`, it will use the virtual environment's python.

PyCharm

PyCharm also has nice support for virtual environments. I will be demonstrating it!

End

The end