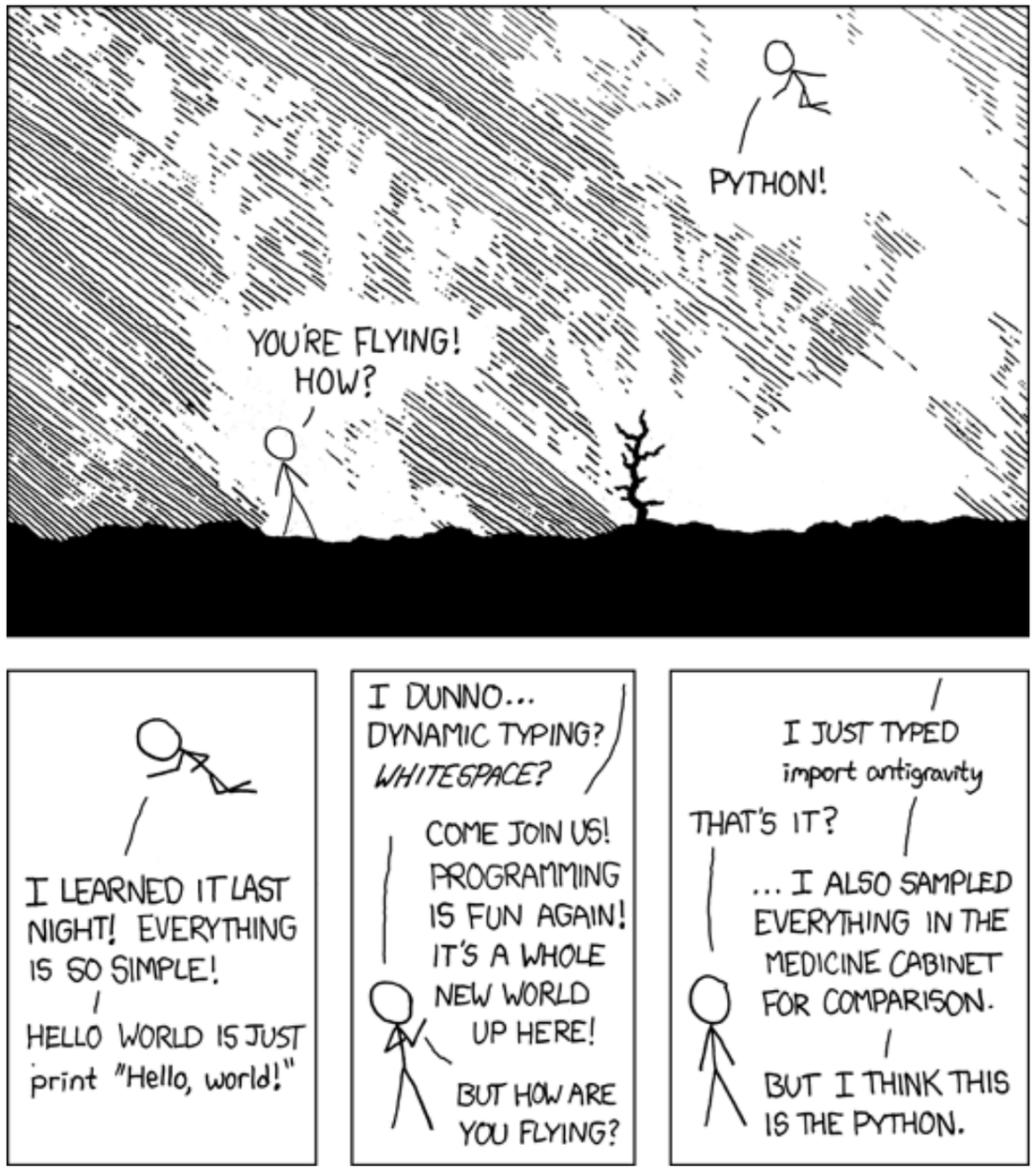
# Getting Started running Python

Virtual environments, installing packages, and running code




https://xkcd.com/353/

Talley Lambert

CITE
Core for Imaging Technology & Education

HARVARD
MEDICAL SCHOOL

# Goals for this Talk

## Understand

1) What Python is
2) Where it "lives" on your computer and where it finds code to run
3) What packages are, and how they bring in additional functionality.
4) What virtual environments are, and how they help us isolate installed packages.

## Know how to

1) Use **uv** to manage python, virtual environments, and packages; and use it to run code.

*We will <u>not</u> be covering how to actually code in python here*

This is an opinionated introduction.

There are *many* different ways to do this stuff.

I'll occasionally provide my personal preference/recommendation

If you already know and like a different approach,
this doesn't invalidate that :)

# What is Python?

"Python is an interpreted, object-oriented, high-level programming language with dynamic semantics."

# What is Python?

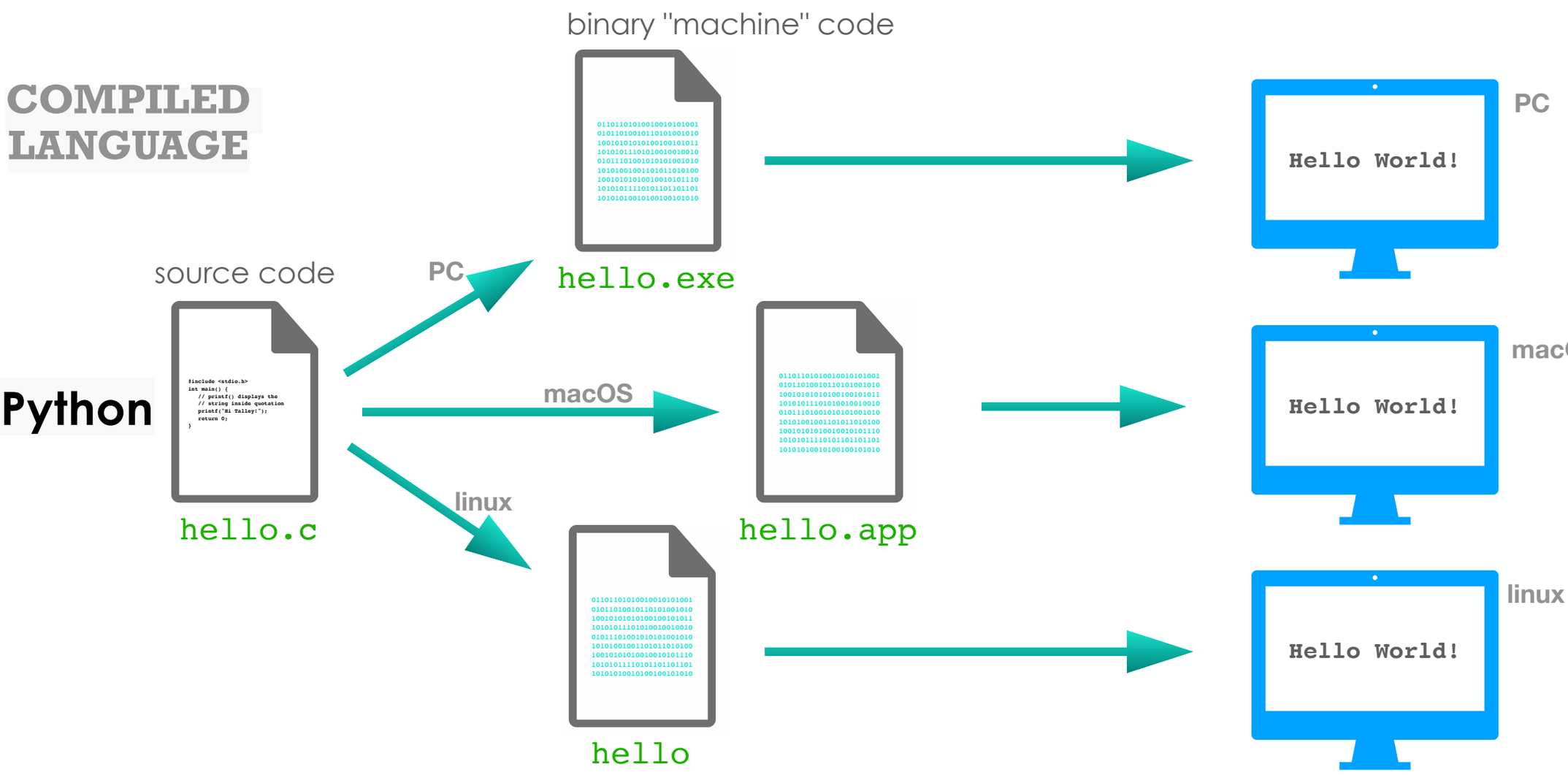
**interpreted**

**Tl;dr:**

With Python, you share and run "regular" text files.
Python does all the OS-specific magic at "runtime".

# What is Python?

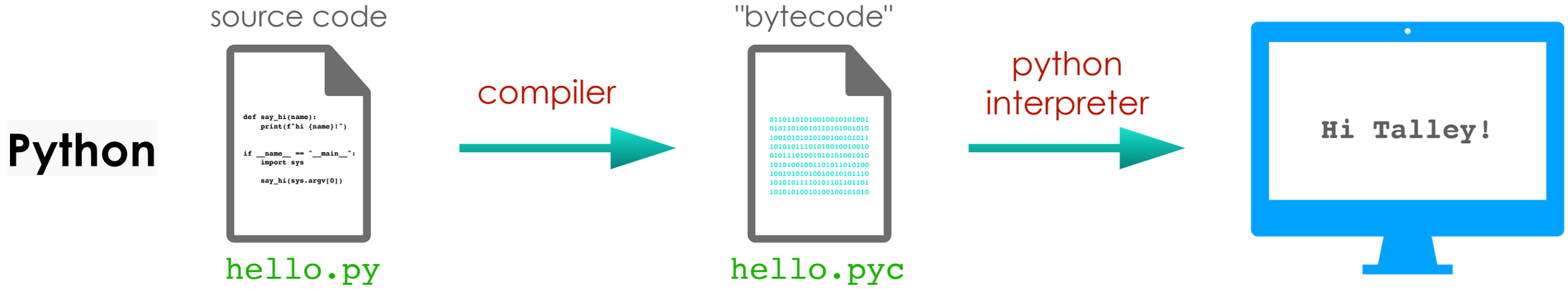
**interpreted**



# What is Python?

**interpreted**

INTERPRETED LANGUAGE




- 👍 Easy **cross-platform** development
- 👍 **Rapid** development
- 👎 Can be slower to execute than compiled languages
  (but can be combined with compiled code if necessary)

https://nedbatchelder.com/blog/201803/is_python_interpreted_or_compiled_yes.html

# What is Python?

**object-oriented**

classes

```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("woof!")

fido = Dog("fido")
```

attributes

methods

instances
("objects")

# What is Python?

## high-level

"high-level" ≅

- easy to understand
- closer to human linguistics
- farther from machine code

👍 Memory management
(you don't have to literally allocate and deallocate memory)

👍 Dynamic typing

👍 Built-in data structures (list, dict, set, tuple, etc...)

https://docs.python.org/3/tutorial/datastructures.html

# What is Python?

## dynamic

```
# make x a string
# no type declaration required!
x = 'hi'

# change it to an integer
# no problem!
x = 2
```

Note: if you *want* types, there are options as well:

https://docs.python.org/3/library/typing.html

# Strengths

## why do scientists use it?

- ✓ Easy to learn, read, and write
- ✓ "Batteries included" (standard library has lots of functionality)
- ✓ Huge community of packages, particularly in data science
- ✓ Rapid development (fast edit-test-debug cycle)
- ✓ Totally free & open-source!
- ✓ "Glue-language" (easy to integrate lower level languages)

# Weaknesses

Faster to develop, but can be slower to execute compared to compiled languages

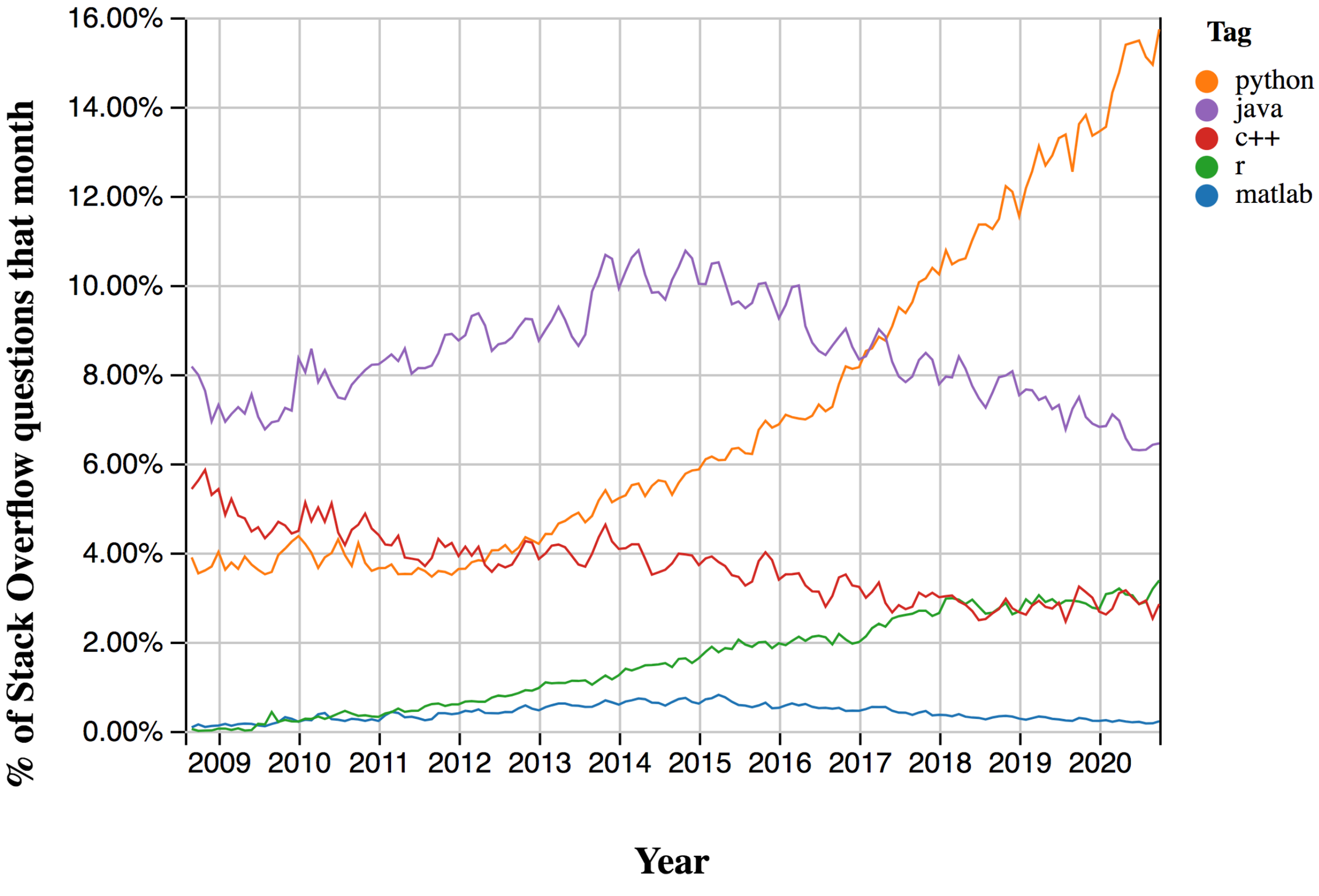
(But... see C-extensions, cython.org, & numba.org, etc...)

Decreased memory efficiency

Parallelism & concurrency requires some workarounds

(But: many solutions exist... see dask.org)

# So hot right now...



# Terms

**python interpreter** — The **program** that actually "runs" (i.e. parses, compiles, interprets) python code. (*ex: "CPython 3.8"*)

**module** — An organizational unit of python code. Usually a **single file** ending in `.py` that contains python definitions and expressions,

**package** — A collection of modules. Usually, this is a **folder** of python modules that also contains an `__init__.py` file. "Package" also frequently refers to an installable python library/application (e.g. numpy, matplotlib, pandas...)

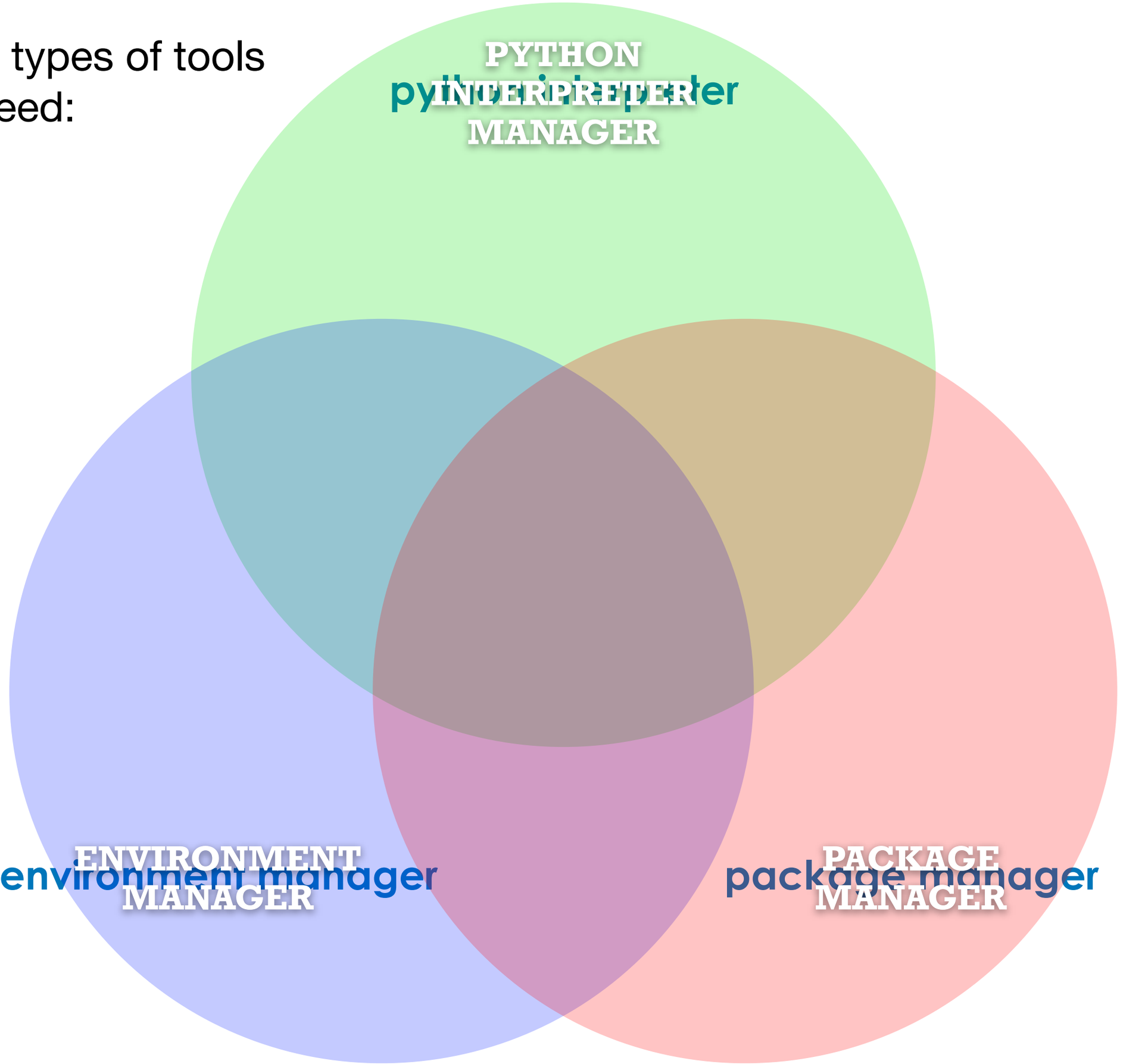
**package manager** — A **program** that automates the installation, updating and removal of packages (e.g. pip, conda)

**virtual environment** — An **isolated collection** of packages, settings, and an associated python interpreter, that allows multiple different collections to exist on the same system

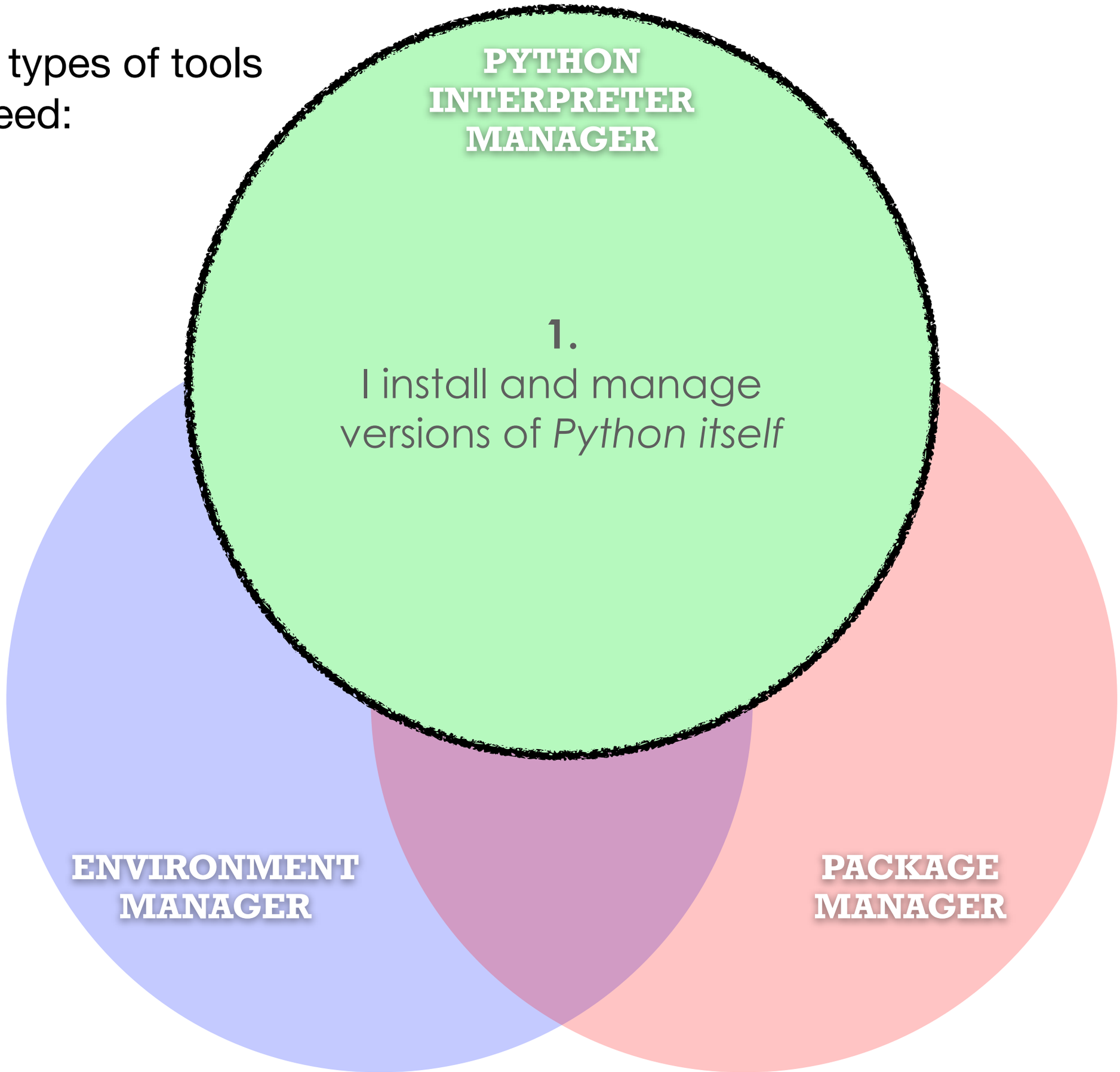
**environment manager** — A **program** that automates the creation and deletion of virtual environments (e.g. conda, virtualenv, venv)

*see also: https://docs.python.org/3/glossary.html*

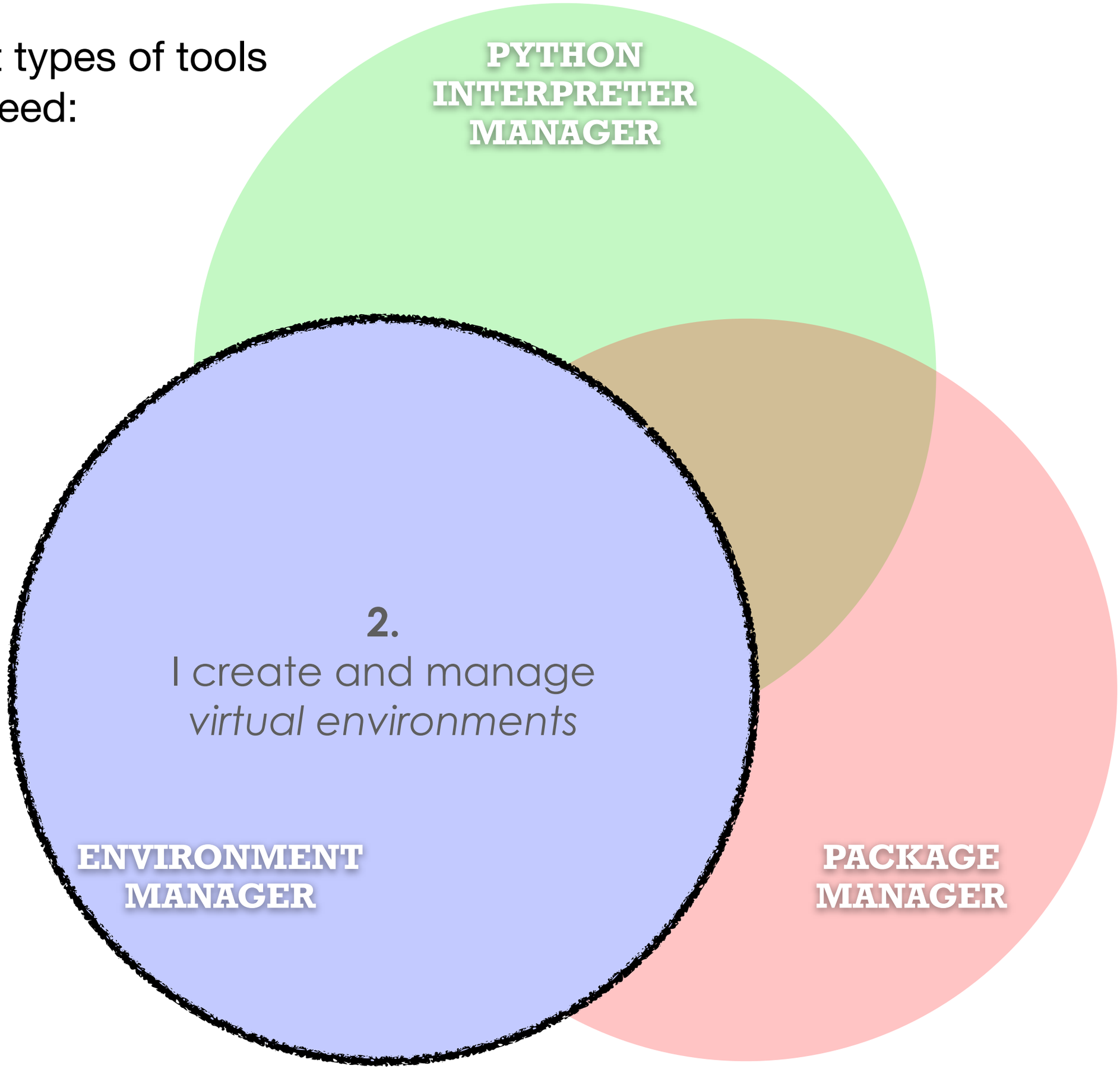
The three important types of tools you will need:



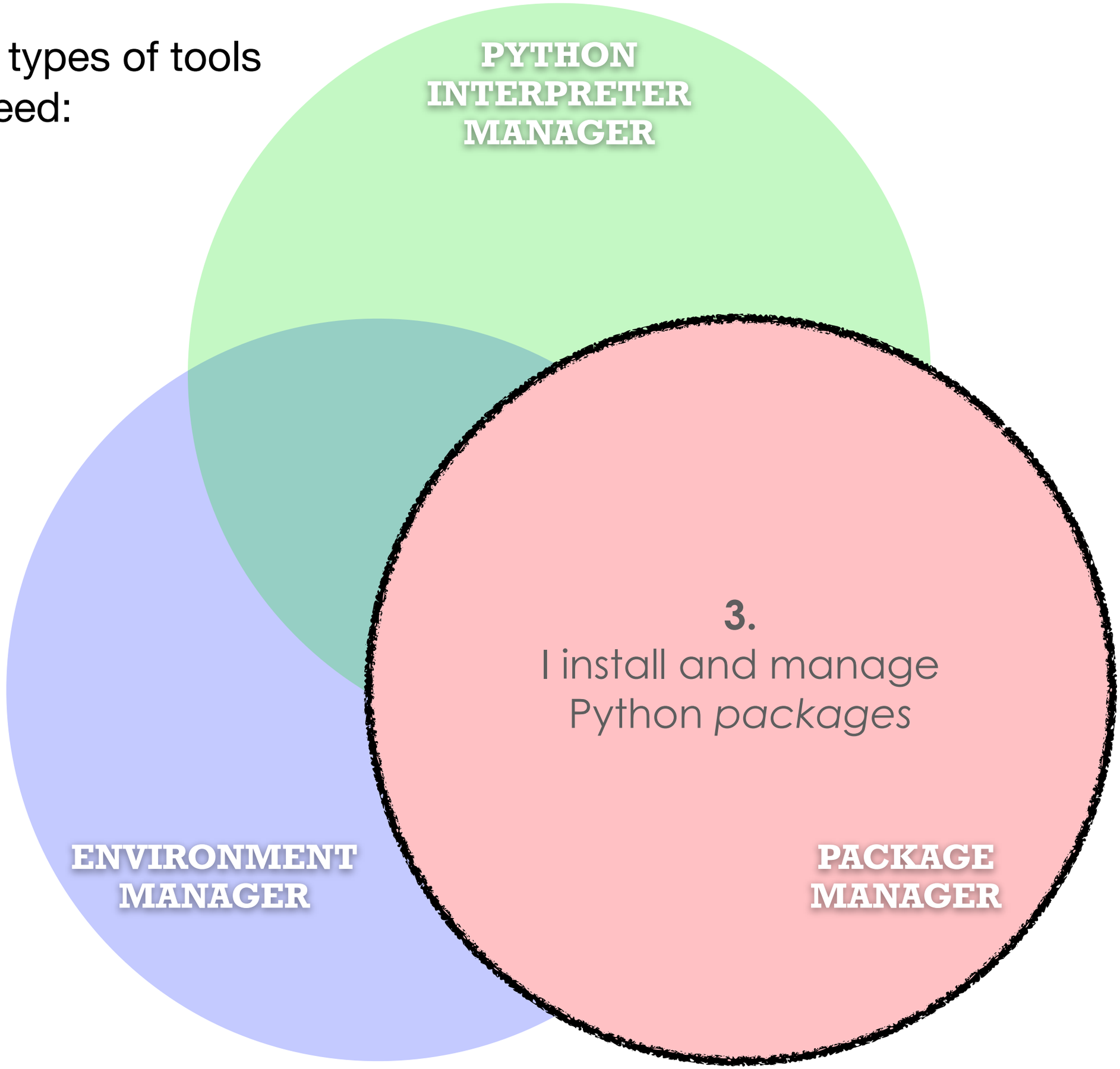
The three important types of tools you will need:



The three important types of tools you will need:



The three important types of tools you will need:



# What does it mean to run code in python?

### Executing a script/program/file

```
# ~/Desktop/hi.py

print("hello world!")
```

```
$ python ~/Desktop/hi.py
hello world!
```

### Using an interactive console or notebook

```
$ python
Python 3.13.5 (main, Jul  8 2025,
20:55:53) [Clang 20.1.4] on darwin

>>>  print("hello world!")
hello world!
```

# Python is modular

One Python module (file) can **`import`** functionality from another module (file)

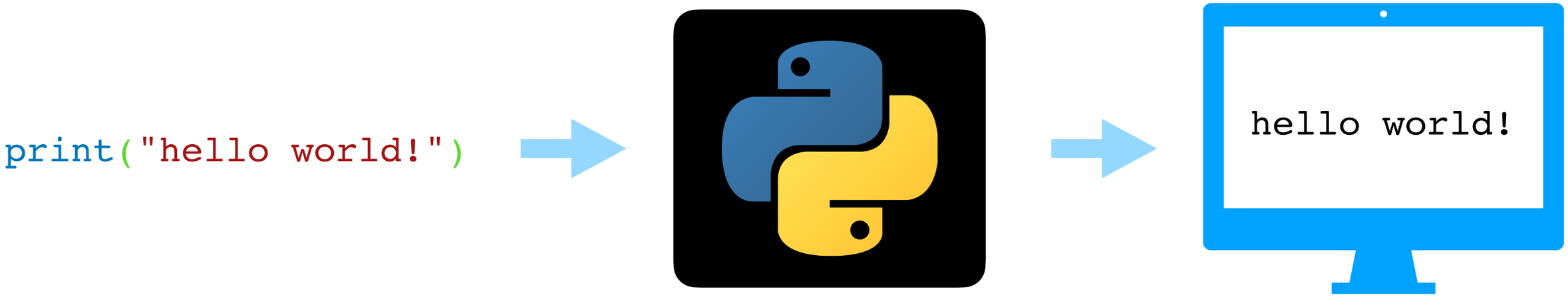
```
# ~/Desktop/do_math.py

from numpy import add

print(add(1, 2))
```

```
$ python ~/Desktop/do_math.py
3
```

# What is the Python interpreter?

```
$ python
```

The **Python interpreter** is a program that parses and compiles source code (human readable code) into "bytecode" (a lower level representation) that is then "interpreted" (executed) by the python "virtual machine"




Source Code

Python interpreter

Magic.

# What is the Python interpreter?

```
$ python
```

The **Python interpreter** is a program that parses and compiles source code (human readable code) into "bytecode" (a lower level representation) that is then "interpreted" (executed) by the python "virtual machine"

# Where does it live?

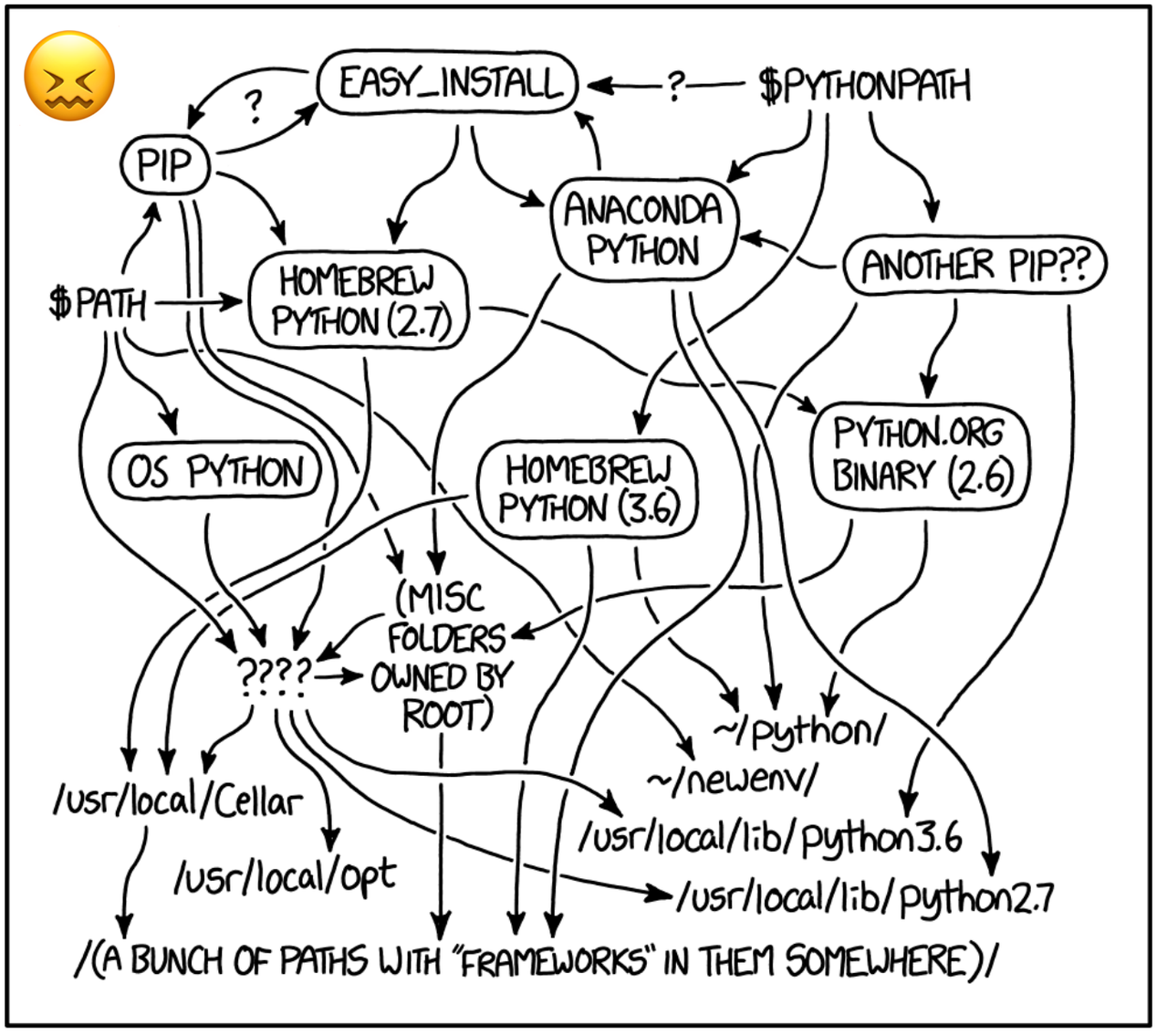
**(It can live almost anywhere)**

```
$ which python
python not found

python is /usr/bin/python  # mac ≤ Catalina 10.15
$ python --version
Python 2.7.16
```

Python 3 has been out since 2008
Python 2 is end-of-life as of Jan. 2020

**For this and many other reasons... don't use your system python (if you have one)!**

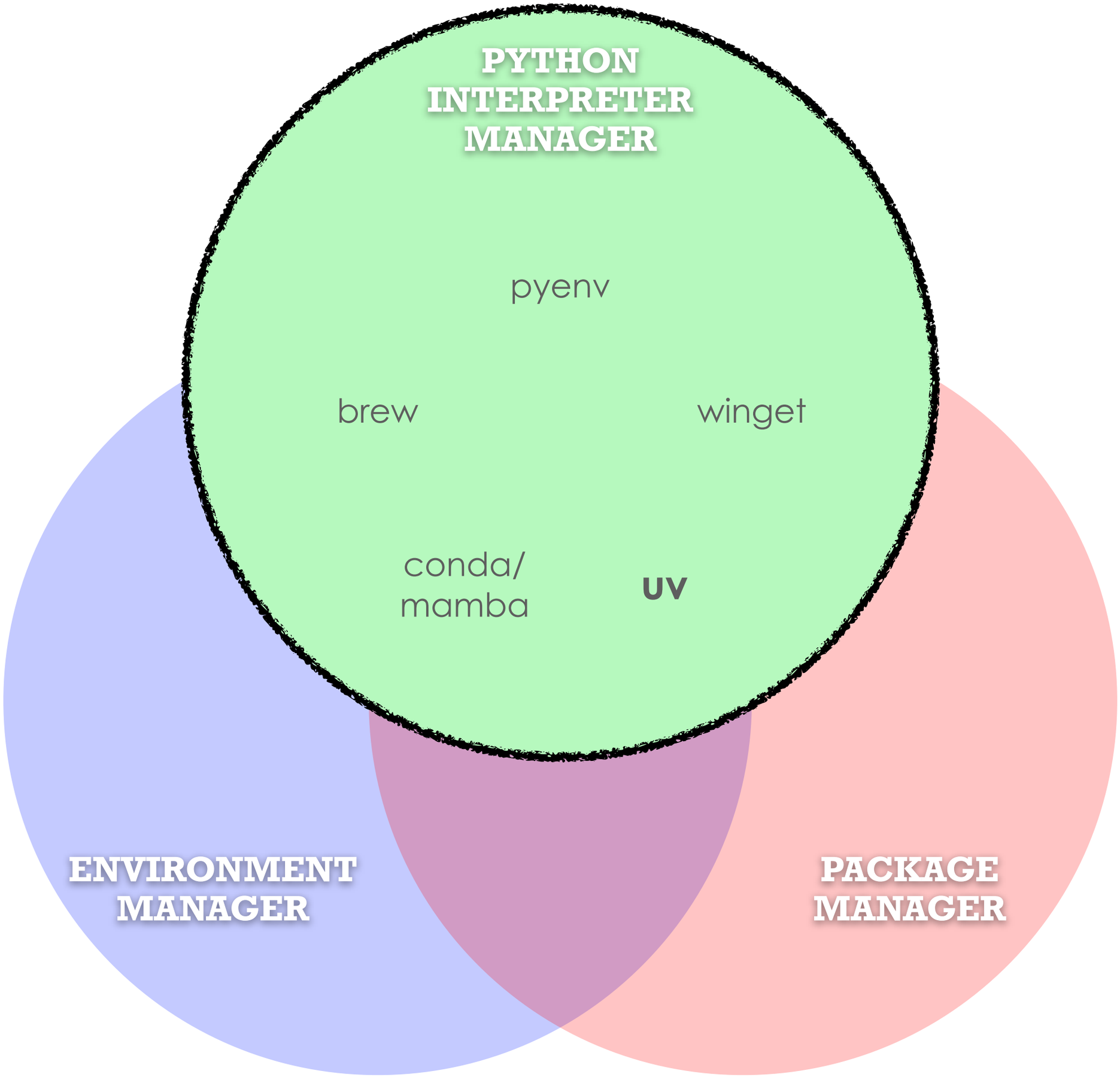
# OK, then how should I get Python?




MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

**TI;dr: install uv**

https://docs.astral.sh/uv/getting-started/installation/


PYTHON INTERPRETER MANAGER
pyenv
brew
winget
conda/ mamba
UV
ENVIRONMENT MANAGER
PACKAGE MANAGER

# Installing uv

Instructions at https://docs.astral.sh/uv/getting-started/installation/

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

# Installing uv

Instructions at https://docs.astral.sh/uv/getting-started/installation/

```
~/Desktop/project
➜ curl -LsSf https://astral.sh/uv/install.sh | sh
```

# Running Python with uv

```
uv run [COMMAND]
```

Ensures that the command or script is run in a Python "environment"
*(more on that in a moment)*.

Will download Python for you if needed.

# What is a python virtual environment?

An isolated collection of **packages**, **settings**, and a **python interpreter**,
... that allows multiple different collections to exist on the same system

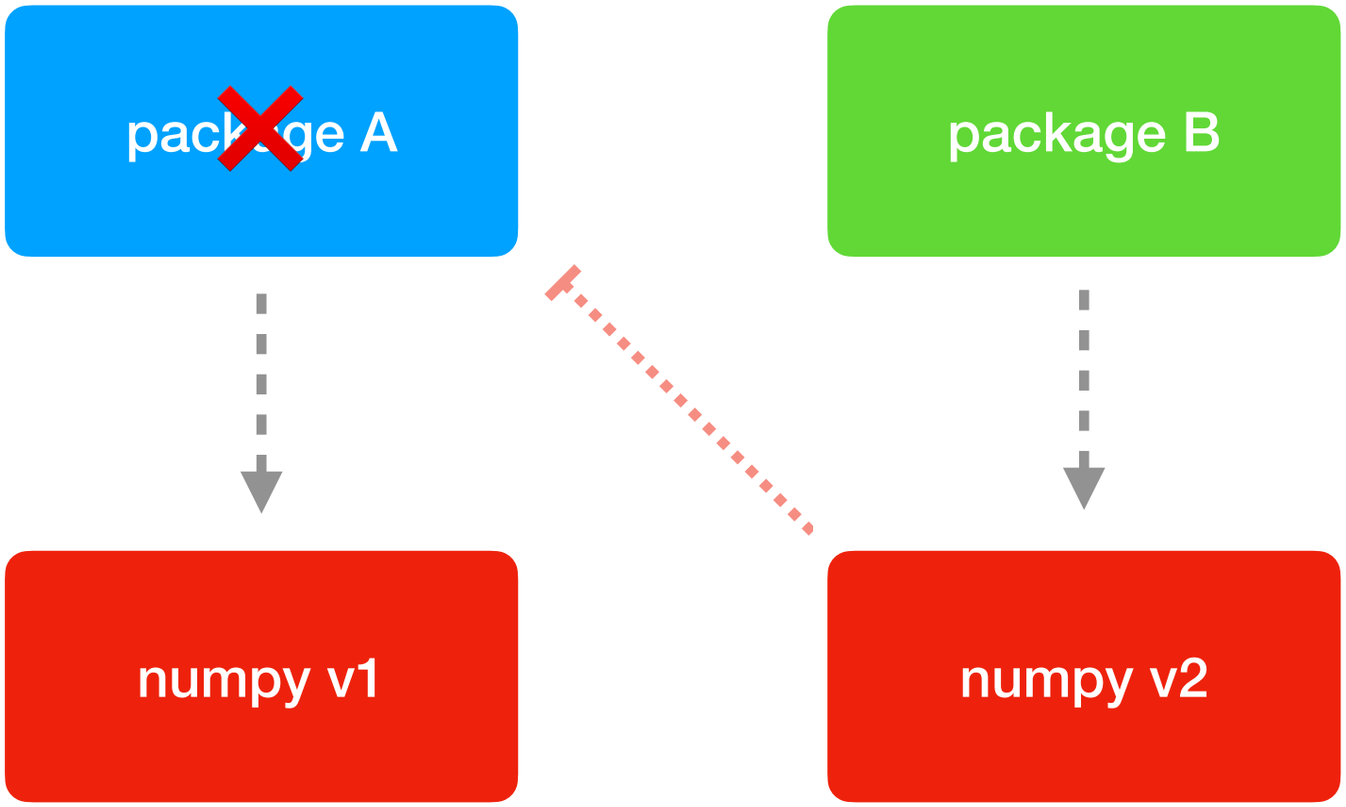
... (It's usually a literal folder somewhere on your computer...)

```
/Users/talley/
├── ...
└── some_folder/
    ├── envA/
    │   ├── lib/
    │   │   └── python3.13
    │   ├── bin/
    │   └── ...
    ├── envB/
    │   ├── lib/
    │   │   └── python3.9
    │   ├── bin
    │   └── ...
    └── ...
```

for more: https://realpython.com/python-virtual-environments-a-primer

# Why would I need more than 1 environment?

**Problem 1:** conflicting package dependencies



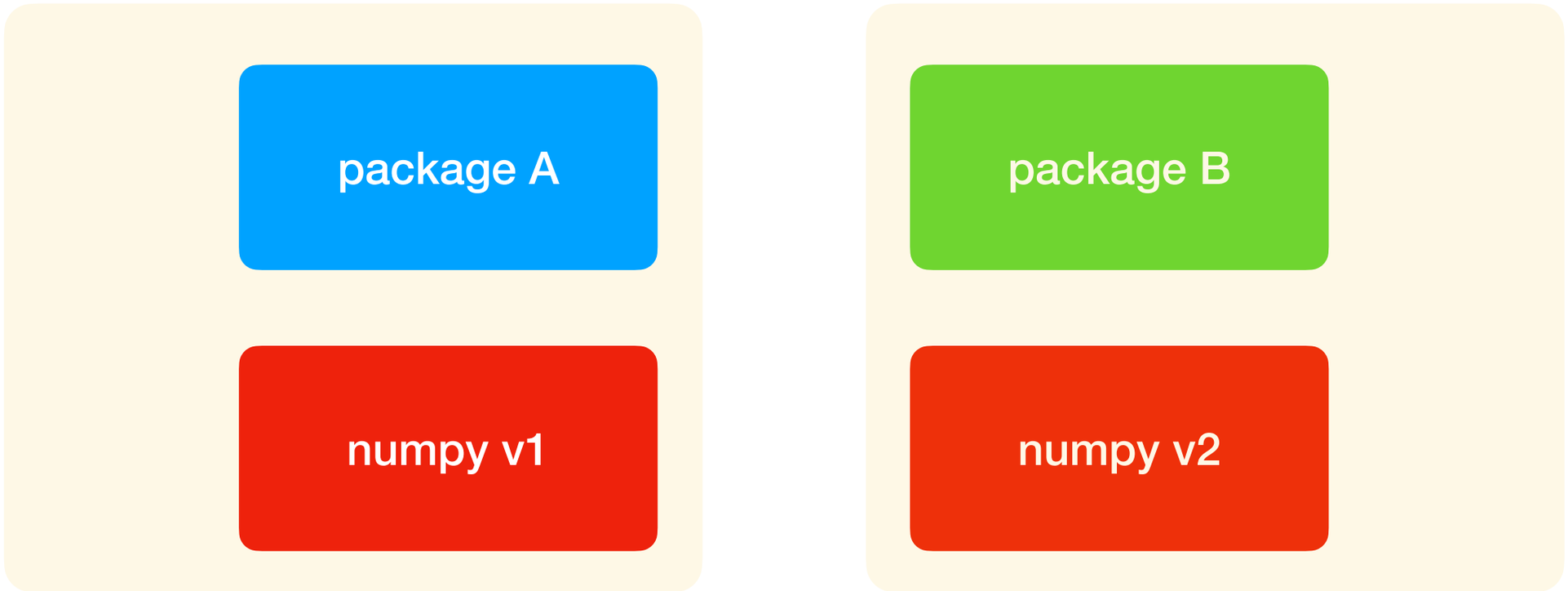

***Tip!*** *you can only have <u>one</u> version of any given package installed in an environment*

# Why would I need more than 1 environment?

**Solution:**

Create **multiple** "virtual" environments

each one contains all of the dependencies and requirements for a particular application



# Why would I need more than 1 environment?

**Problem 2:** "It worked fine on *my* computer!" 🤔

Using environments also keeps your "globally" installed packages to a minimum, and makes your project dependencies much clearer

# Why would I need more than 1 environment?

**Try to avoid having one big "base" environment where you install everything!**

**Think of environments as disposable!**

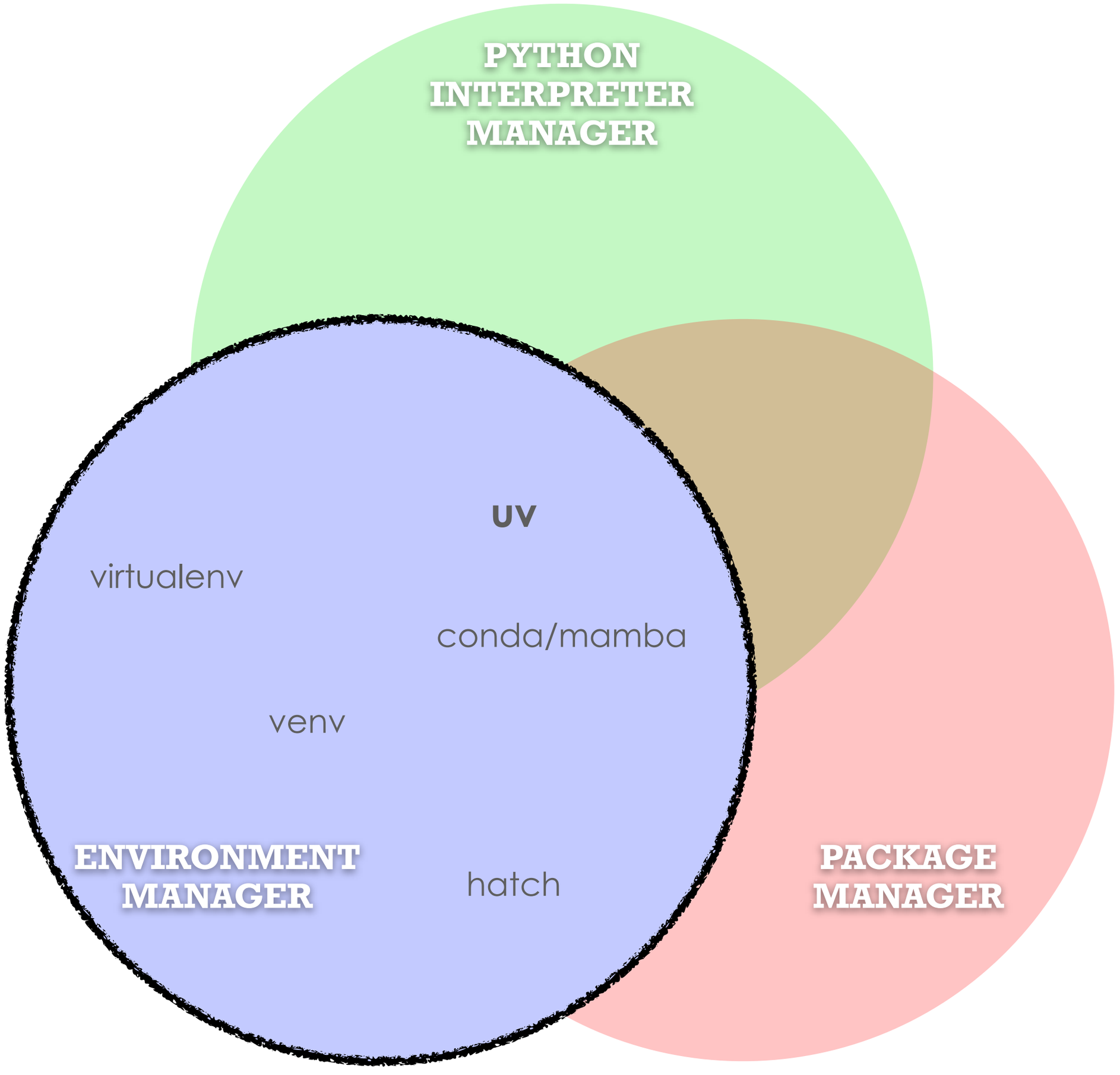
**Don't get "attached" to an environment...
Learn to recreate them quickly**

# How do I create a virtual environment?

You need... **an environment manager**

*Examples*:

- venv
- virtualenv
- conda/mamba
- pipenv
- poetry
- uv


PYTHON
INTERPRETER
MANAGER
UV
virtualenv
conda/mamba
venv
hatch
ENVIRONMENT
MANAGER
PACKAGE
MANAGER

# How do I create a virtual environment?

`uv venv [ENV_NAME]` (defaults to ".venv")

# How do I create a virtual environment?

```
uv venv [ENV_NAME]
```

(defaults to ".venv")

TIP: Specify Python version with `--python/-p`

```
uv venv -p 3.10

uv venv -p 3.13
```

# You need to "activate" it first...

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

```
.venv\Scripts\activate
```

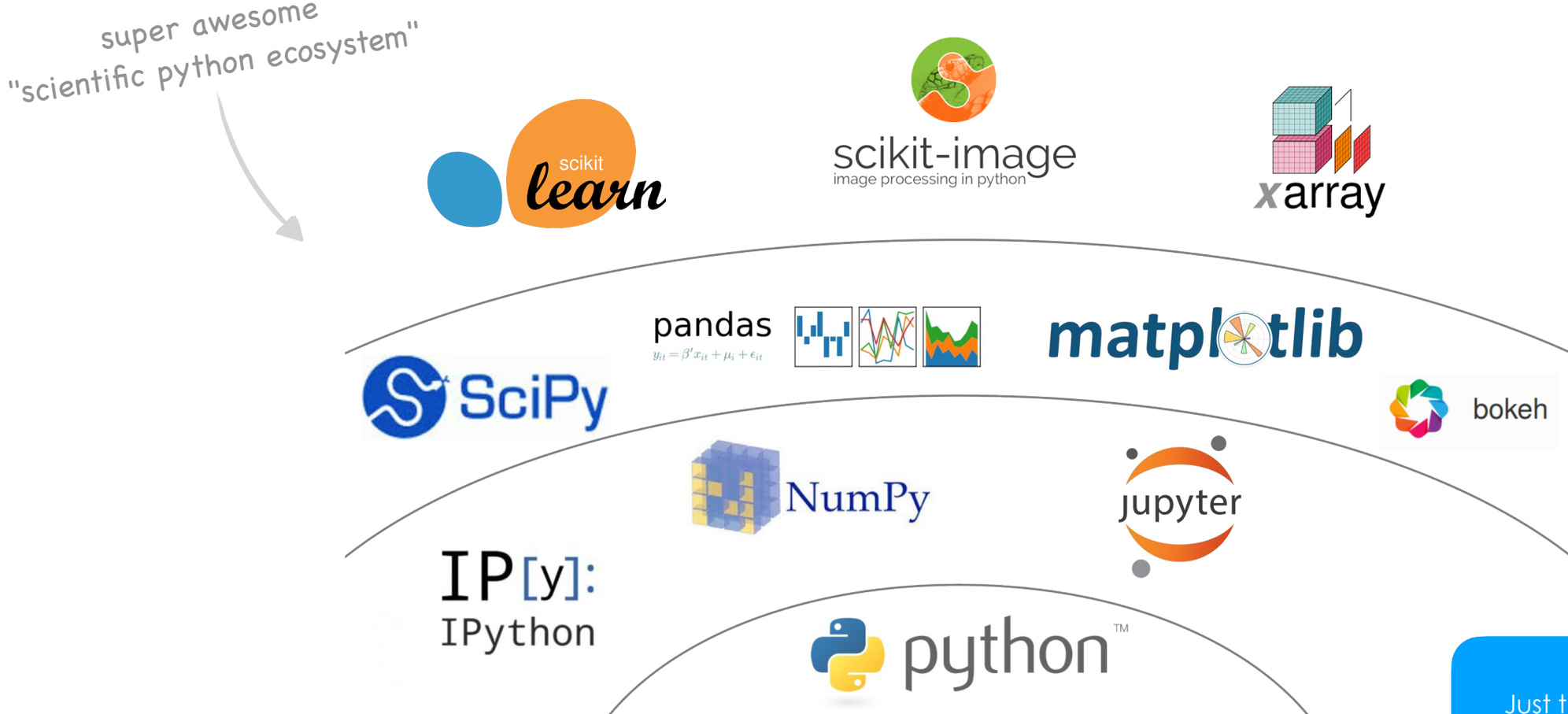
This sets up your terminal environment to know where to find things...

Now that we've got an environment set up...

let's install some packages!

# "Packages" bring in additional functionality

Packages may sometimes be referred to as "libraries"



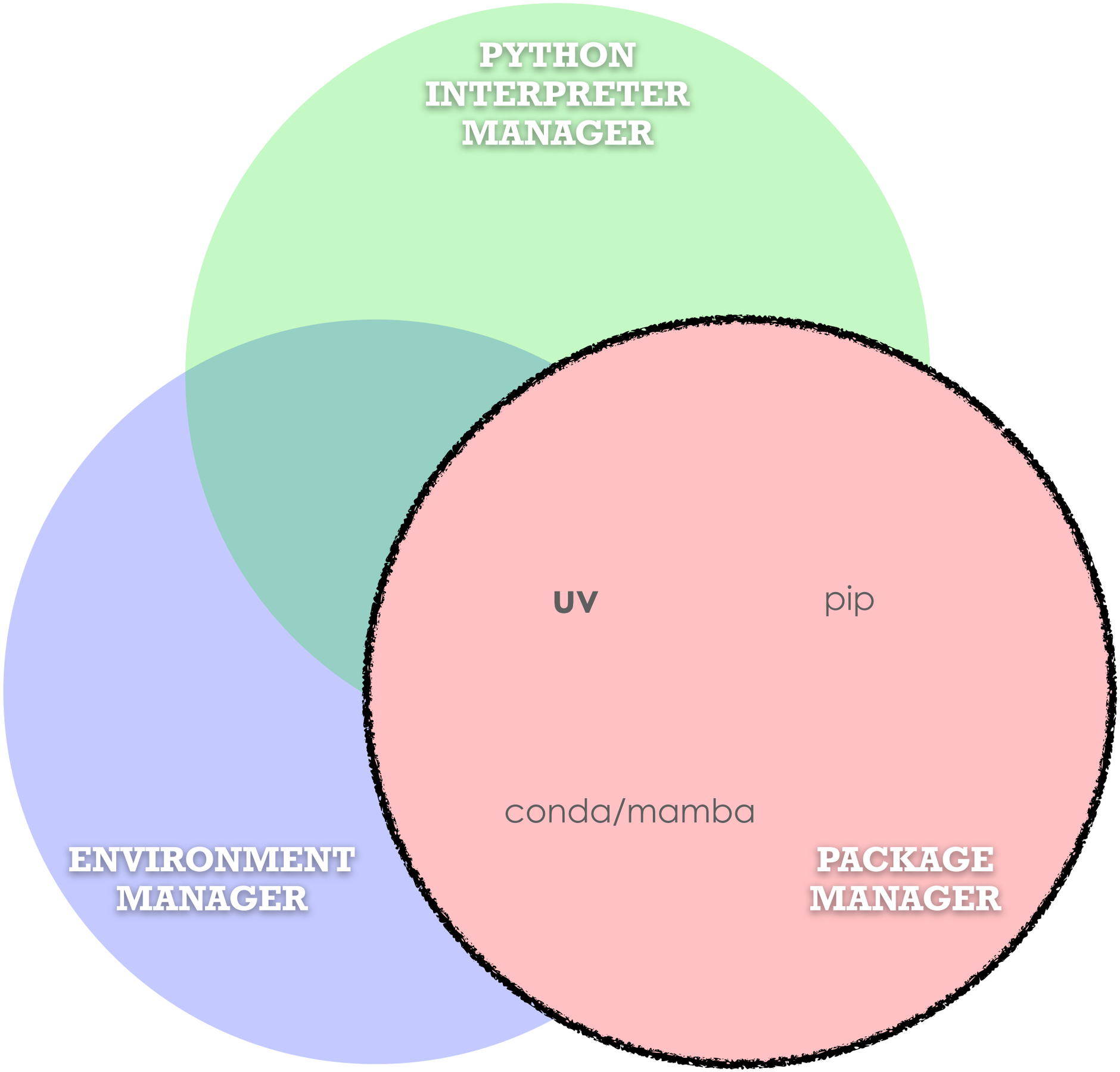

Just think of each of these as a **folder** of **`.py`** files

# How do I get Python packages?

You need... **a package manager**

*Examples*:

- pip
- conda/mamba
- uv pip


PYTHON
INTERPRETER
MANAGER
UV
pip
conda/mamba
ENVIRONMENT
MANAGER
PACKAGE
MANAGER

# How do I get Python packages?

`uv pip install <package>`

```
~/Desktop/project via my_env
→ uv pip install 'cellpose[gui]'
```

# Tip: show installed packages with

`uv pip list`

```
→ uv pip list
Package                 Version
----------------------- ---------
cellpose                4.0.6
fastremap               1.17.1
filelock                3.18.0
fill-voids              2.1.0
fsspec                  2025.5.1
imagecodecs             2025.3.30
jinja2                  3.1.6
markupsafe              3.0.2
mpmath                  1.3.0
natsort                 8.4.0
networkx                3.5
numpy                   2.3.1
opencv-python-headless  4.11.0.86
pillow                  11.3.0
roifile                 2025.5.10
scipy                   1.16.0
segment-anything        1.0
setuptools              80.9.0
sympy                   1.14.0
tifffile                2025.6.11
torch                   2.7.1
torchvision             0.22.1
tqdm                    4.67.1
typing-extensions       4.14.1
```

# Where did it install??  How does Python find code?

```
import os
import sys

import matplotlib
import numpy as np

import my_module
```

**From where??**

# How does Python find code?

`sys`
is a module
in the "standard library"

`sys.path`

`path`
is a variable
in the `sys` module

```
>>> import sys

>>> print(sys.path)
# on mac/linux
[
 '.../.venv/lib/python3.X',
 '.../.venv/lib/python3.X/lib-dynload',
 '',
 '.../.venv/lib/python3.X/site-packages',
]
```

# How does Python find code?

## sys.path

```
>>> import sys

>>> print(sys.path)
# on mac/linux
[
 '.../.venv/lib/python3.X',
 '.../.venv/lib/python3.X/lib-dynload',
 '',
 '.../.venv/lib/python3.X/site-packages',
]
```

```
>>> import sys

>>> print(sys.path)
# on Windows:
[
 '...\.venv\Lib',
 '...\.venv\DLLs',
 '',
 '...\.venv\Lib\site-packages',
]
```

For more:
https://docs.python.org/3/library/sys.html#sys.path
https://leemendelowitz.github.io/blog/how-does-python-find-packages.html

# How does Python find code?

```
# "standard library" modules
import os
import sys

# mac/linux:
<env>/lib/pythonX.Y
# windows:
<env>\Lib


# installed third-party packages
import matplotlib
import numpy as np

# mac/linux:
<env>/lib/pythonX.Y/site-packages
# windows:
<env>\Lib\site-packages


# your own local stuff...
import my_module

""   # meaning "current directory"
```

sys.path

```
[
 '.../.venv/lib/python3.X',
 '.../.venv/lib/python3.X/lib-dynload',
 '',
 '.../.venv/lib/python3.X/site-packages',
]
```

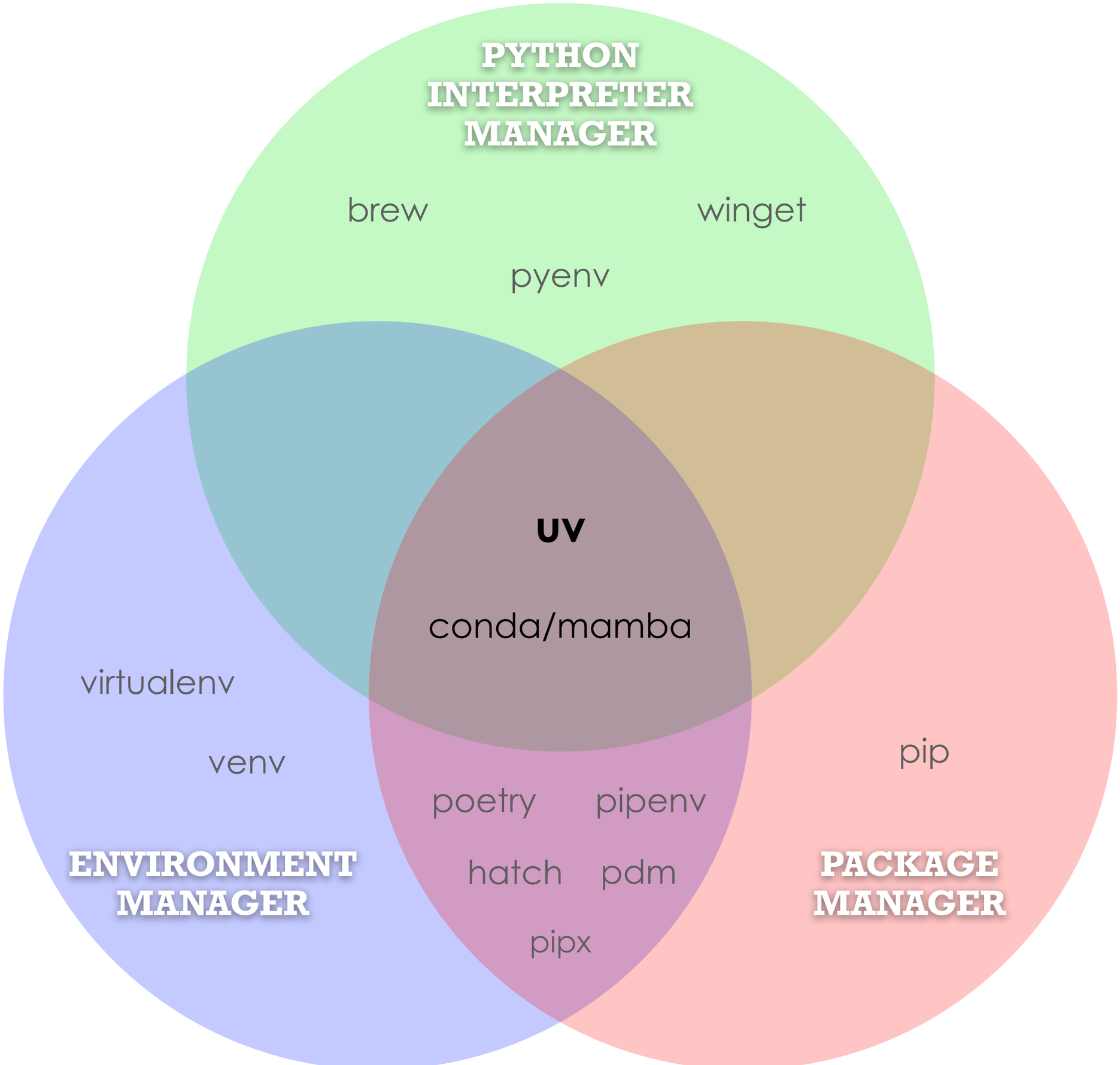
+ anything else you add to sys.path!

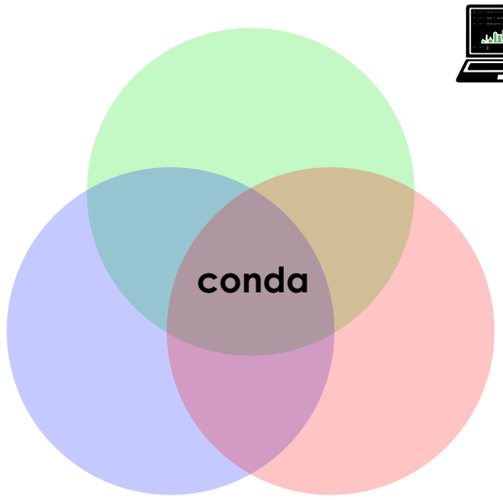
That's where pip installs!

```
# tip; find the file:
import numpy
print(numpy.__file__)
```

# Now you know...

- ☑ What **Python** is
- ☑ Where it "lives" on your computer and **where it finds code** to run
- ☑ What **virtual environments** are, and how they help us isolate installed packages.
- ☑ What **packages** are, and how they bring in additional functionality.


PYTHON INTERPRETER MANAGER
brew
winget
pyenv
UV
conda/mamba
virtualenv
venv
poetry
pipenv
hatch
pdm
pipx
pip
ENVIRONMENT MANAGER
PACKAGE MANAGER




https://github.com/conda/conda

Conda is an open-source, cross-platform, language-agnostic **package management** system and **environment management** system

There are many "flavors" ... I recommend using **micromamba**:

https://mamba.readthedocs.io/en/latest/installation/micromamba-installation.html

For our purposes ... the most important difference between **uv** and **conda** will be where they download packages from

# uv pip vs. conda


https://pypi.org

`pip/uv pip`
installs **Python** packages
from **PyPI**
into **any environment**


https://anaconda.org/

`conda`
installs **any kind of** package
from **Anaconda**
into a **conda environment**

*see also: https://jakevdp.github.io/blog/2016/08/25/conda-myths-and-misconceptions/*

# **uv pip** vs. **conda**

There *may* be cases where you *must* install a package via conda,
e.g. for certain performance GPU computing tasks

... otherwise, stick with `uv venv` and `uv pip`

# (Re)creating environments with dependency files

## pip / uv-pip

requirements.txt

```
numpy>=1.14.1
scipy>=1.0.1
matplotlib>=2.0.0,!=3.0.0
networkx>=2.0
pillow>=4.3.0
imageio>=2.3.0
```

Create env and install

```
uv venv
uv pip install -r requirements.txt
```

https://pip.pypa.io/en/stable/user_guide/#requirements-files

## conda

environment.yml

```
name: myenv
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.7
  - pip
  - numpy>=1.14.1
  - scipy
  - pip:
    - tifffile
```

Create env and install

```
conda env create -f environment.yml
```

https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

# Listing dependencies within a script

my_file.py

```
# /// script
# requires-python = ">=3.13"
# dependencies = [
#     "numpy",
# ]
# ///

from numpy import add

print(add(1, 2))
```

This is a great (newer) convention in the Python world.

It is supported by uv.

```
uv run my_file.py
```

# One more cool trick...

## Run a command provided by *any* Python package

`uvx [COMMAND]`


- Creates a Python environment
- Installs package with the same name as `[COMMAND]`
- Runs that `[COMMAND]`
- If the package and command don't have the same name:

  `uvx --from [PACKAGE] [COMMAND]`

# IP[y]: IPython
## Interactive Computing

https://ipython.readthedocs.io

- Souped up interactive console (use `ipython` instead of `python`)
- Tab autocompletion

```
In [1]: import math

In [2]: math.  <press tab>
acos()      cos()       factorial()  isfinite()  modf()      sqrt()
acosh()     cosh()      floor()      isinf()     nan         tan()
asin()      degrees()   fmod()       isnan()     perm()      tanh()
asinh()     dist()      frexp()      isqrt()     pi          tau
atan()      e           fsum()       ldexp()     pow()       trunc()
atan2()     erf()       gamma()      lgamma()    prod()
atanh()     erfc()      gcd()        log()       radians()
ceil()      exp()       hypot()      log10()     remainder()
comb()      expm1()     inf          log1p()     sin()
copysign()  fabs()      isclose()    log2()      sinh()
```

# IP[y]: IPython
## Interactive Computing

https://ipython.readthedocs.io

- Souped up interactive console (use `ipython` instead of `python`)
- Tab autocompletion
- Easy help/documentation (`?` and `??`)

```
In [5]: math.isclose?
Signature: math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
Docstring:
Determine whether two floating point numbers are close in value.

  rel_tol
    maximum difference for being considered "close", relative to the
    magnitude of the input values
  abs_tol
    maximum difference for being considered "close", regardless of the
    magnitude of the input values

Return True if a is close in value to b, and False otherwise.

For the values to be considered close, the difference between them
must be smaller than at least one of the tolerances.
```

# IP[y]: IPython
## Interactive Computing

https://ipython.readthedocs.io

- Souped up interactive console (use `ipython` instead of `python`)
- Tab autocompletion
- Easy help/documentation (`?` and `??`)
- "magics" (e.g. `%run`, `%debug`, `...`)
  https://ipython.readthedocs.io/en/stable/interactive/magics.html

```
In [1]: %run some_script.py
```

... then resume interactive usage with all of the variables from `some_script.py` available

# IP[y]: IPython
## Interactive Computing

https://ipython.readthedocs.io

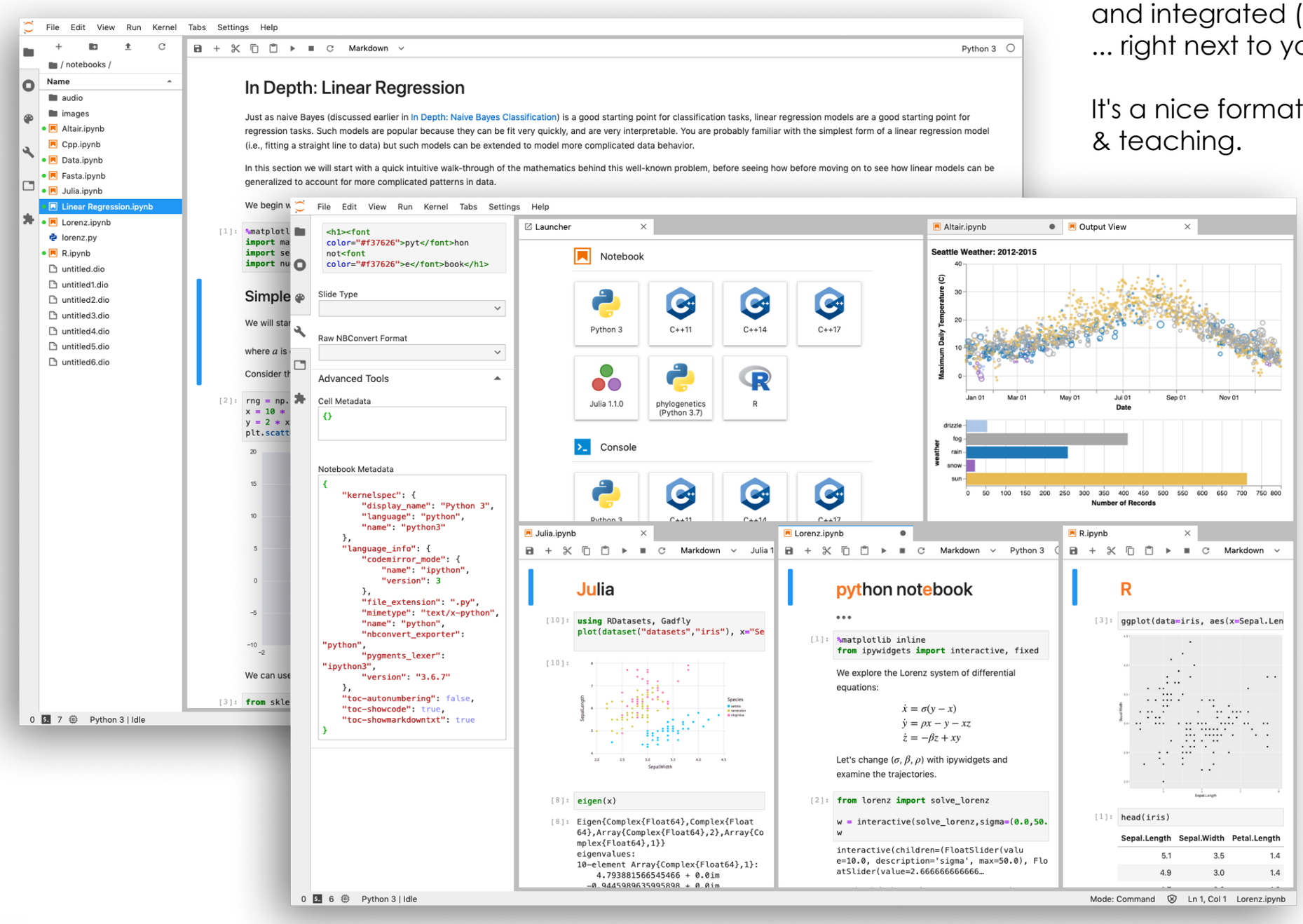
- Souped up interactive console (use `ipython` instead of `python`)
- Tab autocompletion
- Easy help/documentation (`?` and `??`)
- "magics" (e.g. `%run, %debug, ...`)
  https://ipython.readthedocs.io/en/stable/interactive/magics.html
- Command history (press up/down to navigate previous commands)

# jupyter

https://jupyter.org/

# JupyterLab

https://jupyterlab.readthedocs.io/




Jupyter Notebooks offer inline figures and integrated (markdown) documentation. ... right next to your interactive code.

It's a nice format for exploration, communication & teaching.

# Running notebooks with uv via juv

https://github.com/manzt/juv

```
uvx juv init notebook.ipynb
uvx juv run  notebook.ipynb
```

- Ensures a Python environment with Jupyter installed
- Starts a Jupyter server
- Installs any dependencies listed inline at the top of the notebook

Jupyter (notebook/lab) is great

particularly for sharing code with figures & teaching ...

**but**

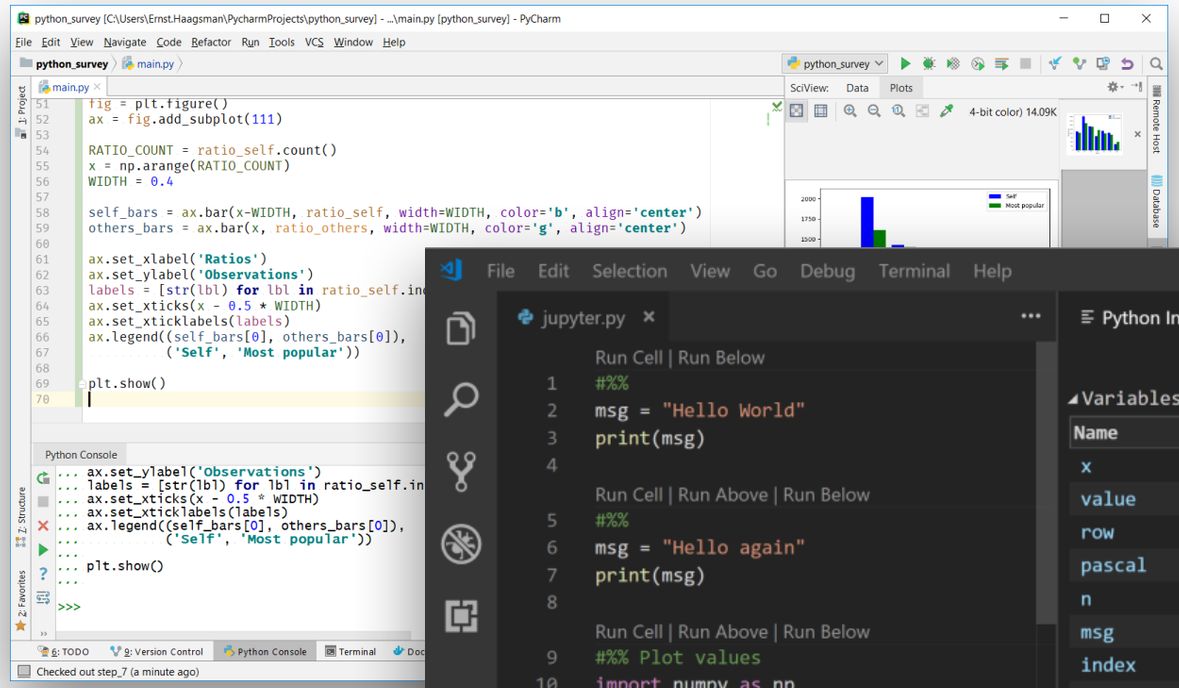
being comfortable using Python/IPython *without* a notebook

is critical

# Integrated Development Environments ("IDE")

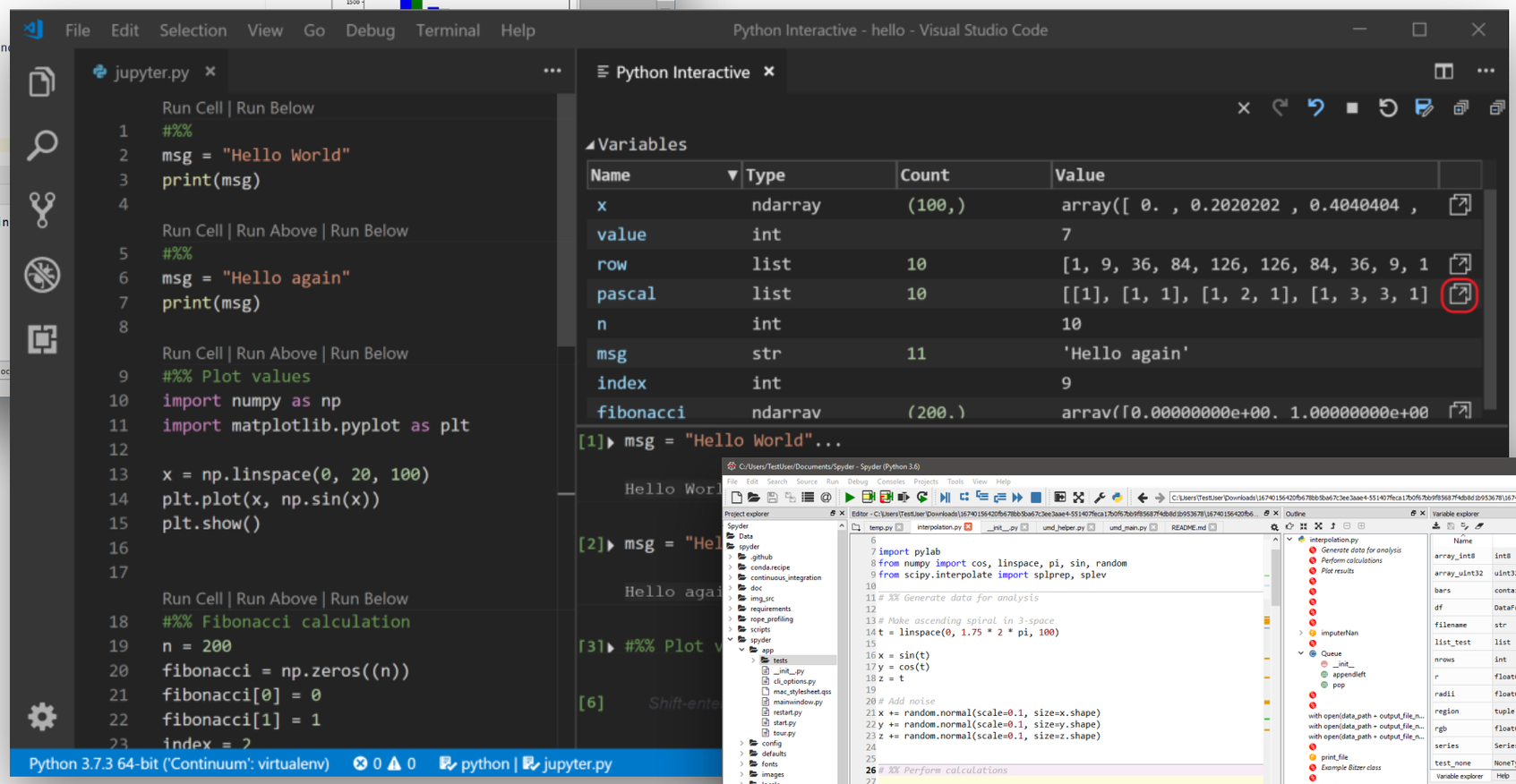
## PyCharm

https://www.jetbrains.com/pycharm/



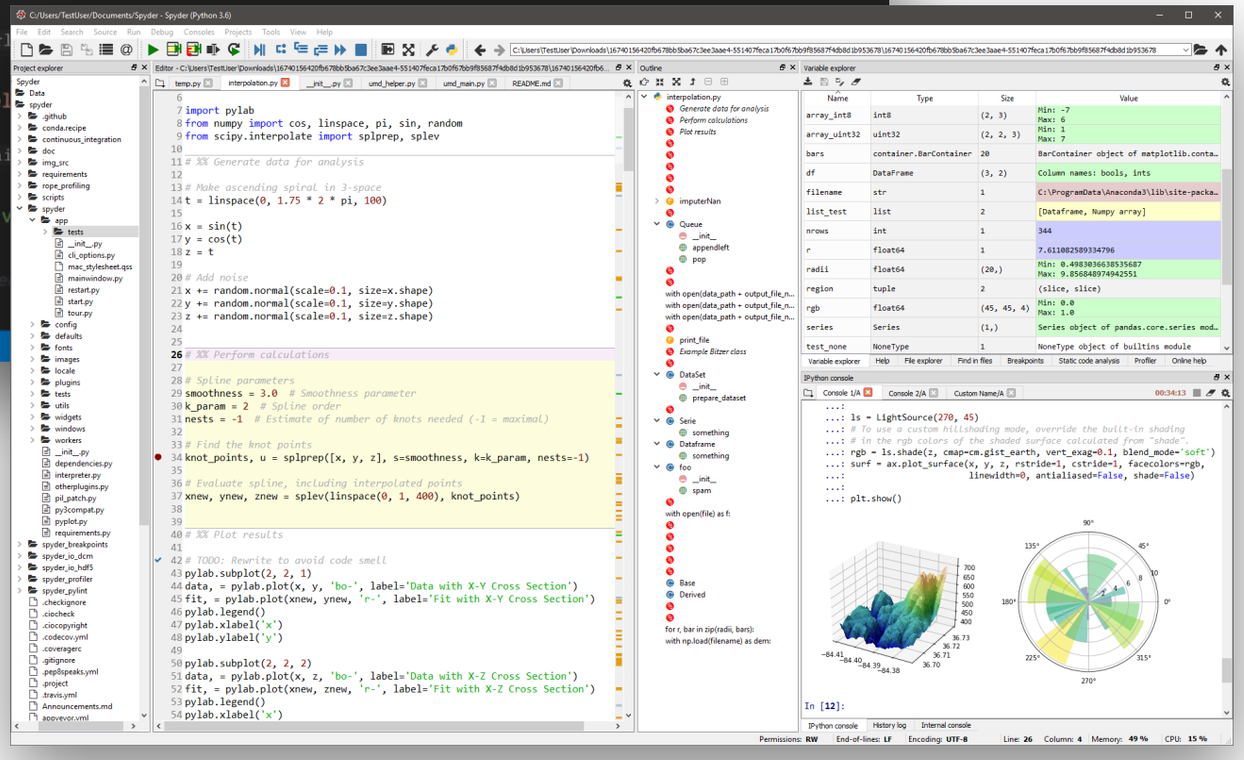

## VS Code

https://code.visualstudio.com/




## Spyder

https://github.com/spyder-ide/spyder



# Summary

- Get Python by installing uv: https://docs.astral.sh/uv/getting-started/installation/
- Create new environments (often) with: `uv venv`
- Consider environments "disposable" ... don't get attached :)
  Use `requirements.txt` (for pip) or `environment.yml` (for conda) files to rebuild them easily
- Use `uv pip install` to install packages.
  (For some difficult binary packages, you'll need conda)
- If you're experiencing mysterious errors... make a new environment!
- `uv run <command>` to quickly execute a script/command in an environment
- `uvx <command>` to run a command from *any* package in an isolated env.
- Use Jupyter for a browser based IDE/notebook/console with rich media, widgets, markdown, etc... But know how to use Python without it!

# Any questions?

# lingering confusions?