LECTURE 02

Computing Fundamentals

September 18, 2023

PBHLTH 198, Fall 2023 @ UC Berkeley

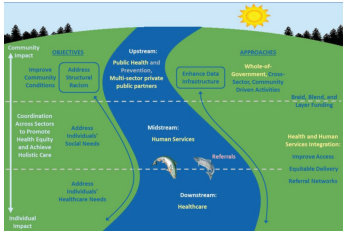
Andrew O'Connor

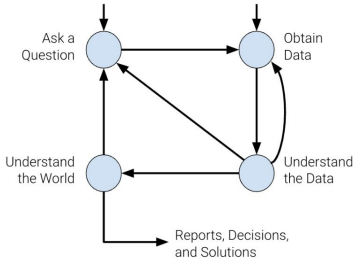
Class Outline

- Recap of lecture 1
- Motivation
- Python fundamentals + terminology
- Command Line
- Dependency Management
- Setup

Recap

- **Public Health** is about disease prevention in order to prevent disease, prolong health, promote health
    - Many ways to do this; Can do it through health policy, community engagement, getting involved with politics, or using statistics and epidemiological theory to help public officials make informed decisions to ultimately eliminate disease (this class!)
- **Epidemiology** seeks to understand disease within populations and design studies to explore the how, why, and where of disease occurrence
- **Biostatistics** aims to analyze health data using advanced statistical methods to derive valid conclusions
- **Data analysis frameworks** provide a structured and efficient approach to data analysis, ensuring consistency and quality in results while facilitating collaboration and compliance with regulations regarding privacy
    - Examples: **PPDAC**, **Data Science Lifecycle** (we will focus on this more)





Motivation

Main goals of this class

- Understand how to install Python locally for the purpose of data analysis in a Jupyter Notebook
- Understand the concept of dependency management and

During class, we ran into difficulties setting this up on some devices
As long as you are able to run code in a Jupyter Notebook (using any method), feel free to continue doing that
At the very least, you are exposed to the concept of managing dependencies and why it is important

Motivation - Workflow

1. Open up the command line
2. Activate a virtual environment

   ```
   conda activate <name of virtual environment>
   ```

3. Install necessary libraries using the command line

   ```
   conda install -c anaconda numpy
   conda install -c conda-forge matplotlib
   conda install -c anaconda pandas
   conda install -c anaconda seaborn
   ```

4. Open JupyterLab

   ```
   jupyter-lab
   ```

5. Open an existing Jupyter Notebook
6. Run cells, do data analysis, etc.
7. Document the changes made to the project (ignore for now)

Terminology

- **Software:** Applications or programs on your computer (e.g. Excel, Windows 10, Zoom)
- **Hardware:** Any physical part of a computer (e.g. CPU, GPU, RAM, hard drive)
- **Machine**: Computer or device capable of doing various tasks, computation, data processing; Can be physical or virtual
- **Local/Locally**: Referring to your own computer
- **Environment**: Refers to a specific type of setup or configuration (includes software, hardware, libraries, packages, etc.)
- **Operating System**: core software that manages hardware resources and provides a platform for running applications; controls tasks like file management, memory allocation, and user interface, allowing users and software to interact with the computer's hardware
    - examples: MacOS, Linux, Windows

Python

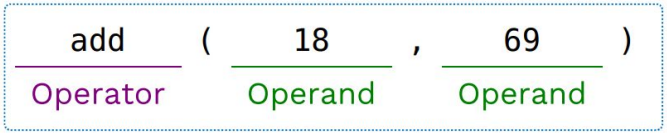
What do programs do?

- **Programs** work by manipulating values
- **Expressions** in programs evaluate to values
  - Expression: `'a' + 'hoy'`
  - Value: `'ahoy'`
- Every value has a certain **data type**

Data type	Example values
Integers	`2` `44` `-3`
Floats	`3.14` `4.5` `-2.0`
Booleans	`True` `False`
Strings	`'¡hola!'` `'its python time!'`

Expressions (Two Ways)

- Expressions can either use **operators** or **function calls** to obtain a value
- **Operators**: special symbols or keywords that are used to perform various operations on variables and values
    - Examples: + (addition), / (division), * (multiplication), ** (exponentiation), = (assignment)
- **Function Calls**: Pieces of predefined code we can reuse to return a value

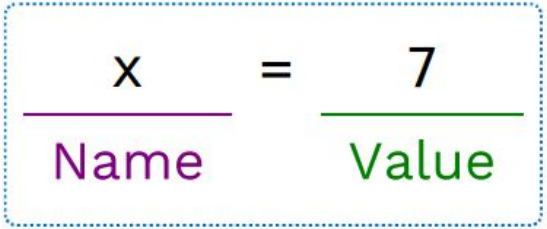


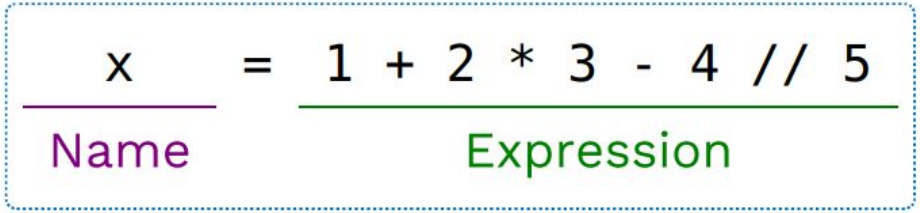
How Python evaluates a call expression:

1. Evaluate the operator
2. Evaluate the operands
3. Apply the operator (a function) to the evaluated operands (arguments)

Names & Variables

- **Names** are bound to a value
  - Values can be expressions
- Names that are bound to values are also called **variables**





Functions

- **Functions** are sequences of code that perform a specific task that can easily be reused
  - The first line of a function is called the **function signature**
  - All lines after are considered the **function body**
  - Functions take in 0 or more **parameters**
  - Some functions have helpful comments that describe what the function does – these are called **docstrings** and they appear on the second line

```
def <name>(<parameters>):          # ← Function signature
    return <return expression>     # ← Function body
```

```
def add(num1, num2):               # ← Function signature
    return num1 + num2             # ← Function body
```

Built-in Functions

- Python has a list of **built-in functions** that can be called at any time without the need of installing/importing libraries

Built-in Functions

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	F	M	
ascii()	filter()	map()	S
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	G	N	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	H	object()	
callable()	hasattr()	oct()	T
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		P	V
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	Z
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		_
	iter()		__import__()

Conditionals

- **Conditionals** are statements that allow a program to make decisions and take different actions based on whether a specified condition evaluates to True or False
    - True and False values are called **booleans**

```
if num < 0:
    sign = "negative"
elif num > 0:
    sign = "positive"
else:
    sign = "neutral"
```

Syntax tips:

- Always start with `if` clause.
- Zero or more `elif` clauses.
- Zero or one `else` clause, always at the end.

```
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
else <condition>:
    <statement>
    ...
```

```
if temperature < 0:
    clothing = "snowsuit"
elif temperature < 32:
    clothing = "jacket"
else:
    clothing = "shirt"
```

Iterables

- In Python, **collections** are containers or data structures that can hold multiple values or objects
    - They are used to group related data and provide convenient methods for manipulation and iteration
- Collections are **iterables** because are objects that can be **looped**/**iterated** over
- List[ ], Tuple( ), Set ( ), Dictionary { }

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`

Loops

- In Python, **collections** are containers or data structures that can hold multiple values or objects
  - They are used to group related data and provide convenient methods for manipulation and iteration
  - collections are considered **iterables** because are objects that can be looped/iterated over
- Two main types of loops
  - **For loops**
  - **While loops**

```
for i in range(10):
  if i == 5:
    break
  print(i)
```

Break Statement

Body of the For Loop

```
0
1
2
3
4
```

Output

While loop used as we don't know the number of times to repeat the loop

```
while True:
    user_input = input("Please enter a number: ")
    if user_input.isdigit():
        print("Thank you!")
        break
    else:
        print("Invalid input. Try again.")
```

```
Please enter a number: r
Invalid input. Try again.
Please enter a number: 2
Thank you!
```

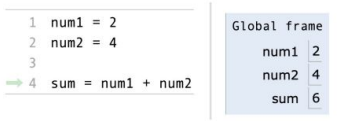
Invalid input

Valid input

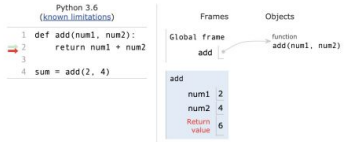
Environments

- All Python code is evaluated in the context of an **environment**
  - An environment is a sequence of **frames**
- **Global frame**: the top-level scope of your program, where variables and functions defined at the highest level are stored
  - Variables declared in the global frame are accessible from anywhere in your program
- **Local frame**: the local scope of a function
  - contains information about the function's parameters, local variables, for the execution of the function

Global frame



Function's local frame, child of Global frame

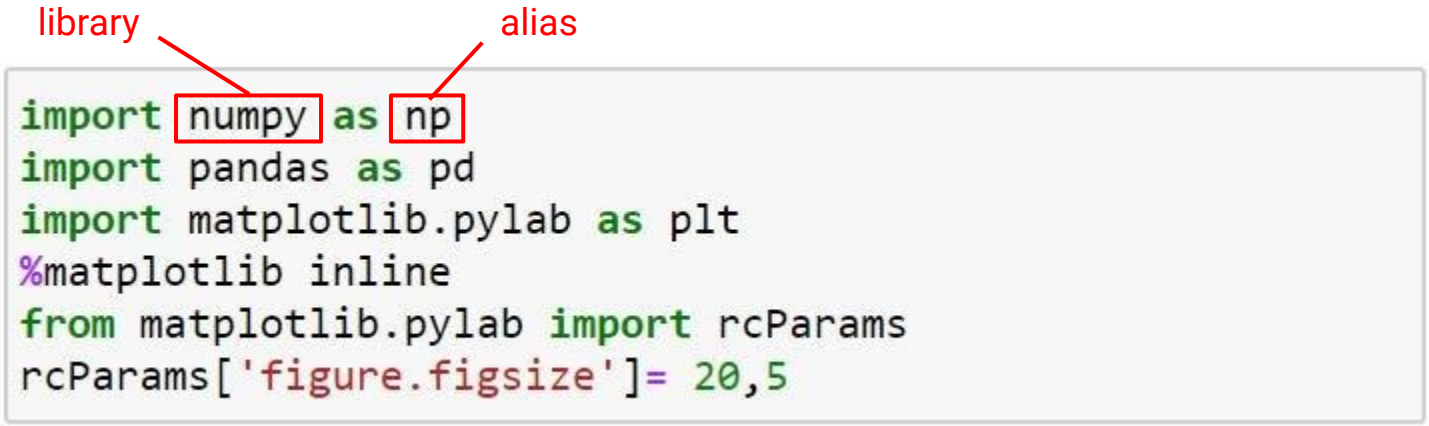


Python Libraries

- **Python libraries** are collections of pre-written code that provide various functions, classes, and modules to help developers and analysts perform tasks without reinventing the wheel
- Common use cases for libraries
    - Data manipulation, Machine learning, Game development, Web development, Data visualization
- Python has a **standard library** where no installation is necessary, you just need to import them
- Other external libraries need to be installed using **package management tools**
    - Created by organizations, open-source, updated frequently

Importing Libraries

- To use functions from existing libraries we need to **import** them into our environment
  - Often, we shorten the name of a library to an **alias** so that it makes calling functions from the libraries easier



Function Calls From Libraries

- When using functions from external libraries, we often use **dot notation** when calling functions

python lists	numpy arrays	
`a.sort()`	`a.sort()`	sorts in-place
`sorted(a)`	`np.sort(a)`	returns new sorted array
`a.sort(key=f)`	–	key function
`a.sort(reversed=False)`	-	ascending/descending

python list vs numpy array

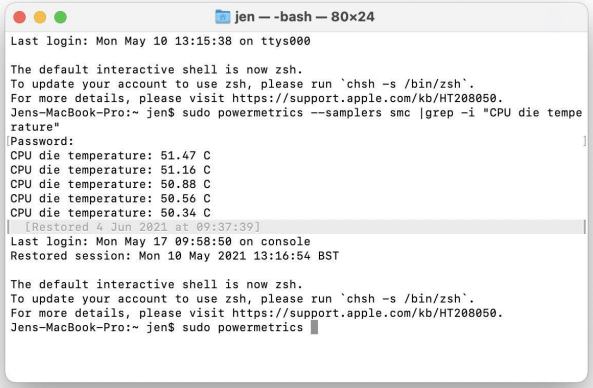
```
a = [1, 2, 3]
b = a          # no copy
c = a[:]       # copy
d = a.copy() # copy
```

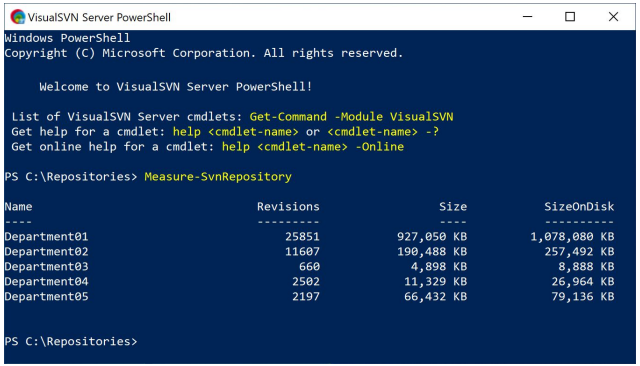
```
a = np.array([1, 2, 3])
b = a          # no copy
c = a[:]       # no copy!!!
d = a.copy() # copy
```

Command Line

Command Line

- For the purposes of this class, we won't go far into the details your computer's operating system or how it works
  - It's very complicated!
- Just know, in data science it is often used to
  - Make/Delete files/folders
  - Launch JupyterLab sessions
  - Interact with GitHub (using Git)
  - Install libraries/packages
- **Command line** is the broader concept of communicating with your computer's operating system but you will often hear "Terminal", "Shell", "Powershell" being used





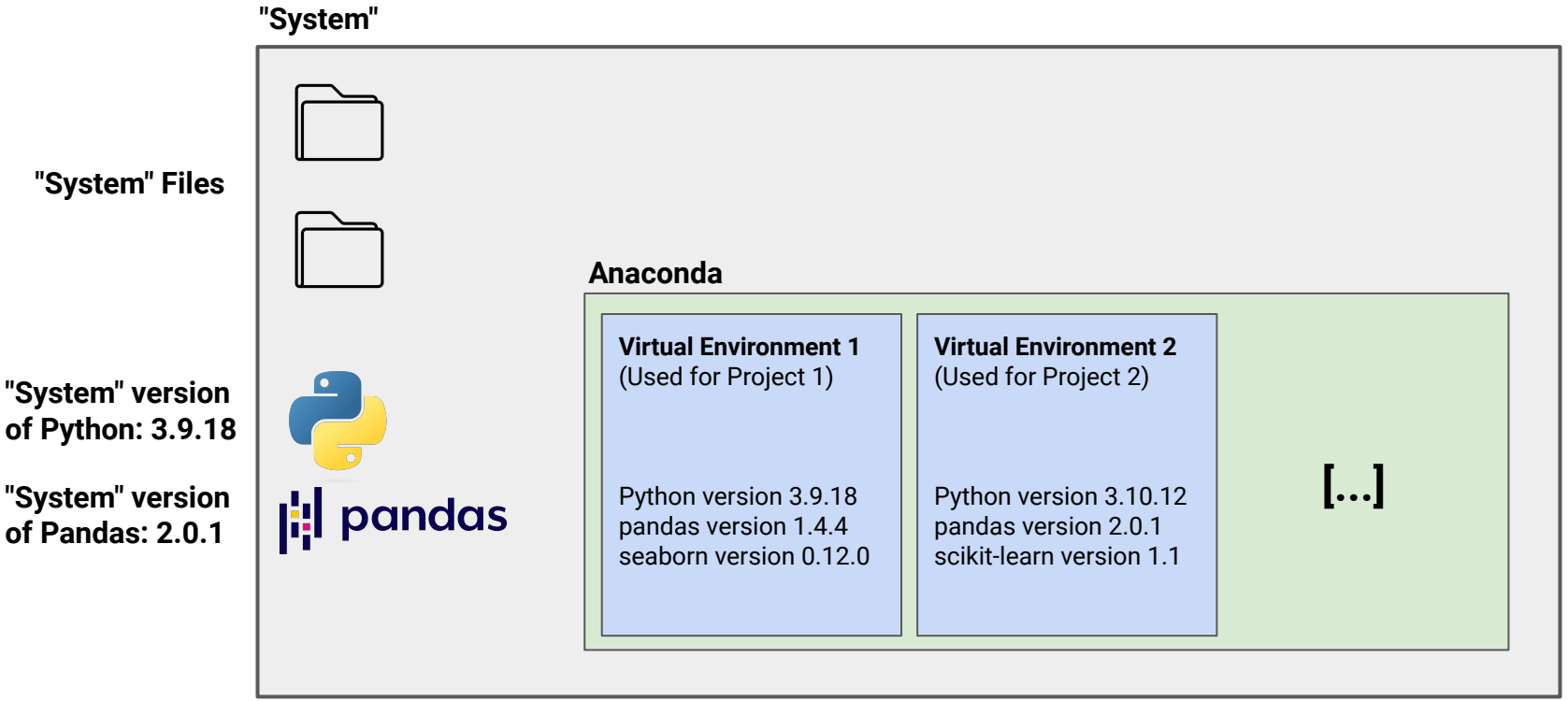
Dependency Management

Managing Projects

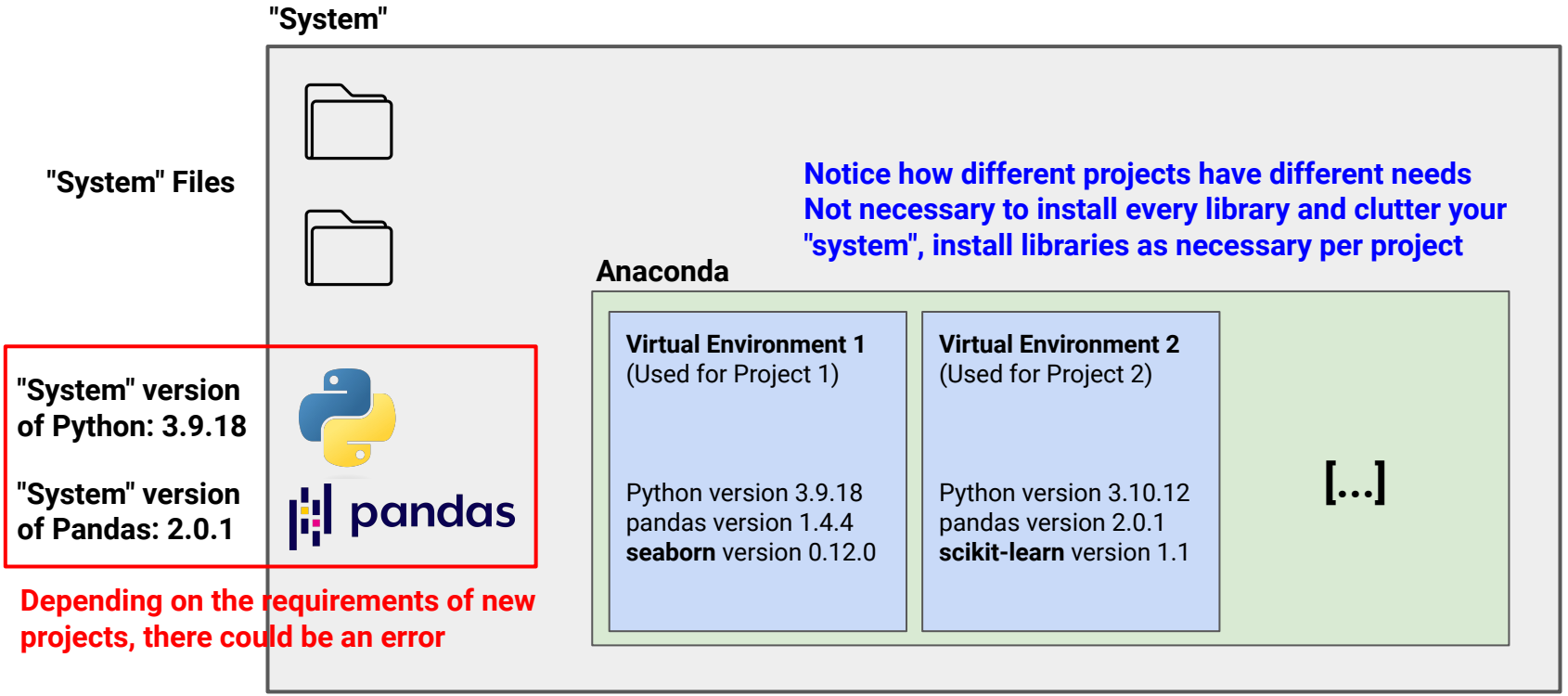
- **Dependency management** is the practice of handling external libraries, packages, and modules that your project relies on
    - Often as a data professional, you will work on different projects at different times – each with their own versions of libraries, configurations, etc.
- Not properly managing dependencies between projects will leave you with **version conflicts**, inconsistent computing environments, inconsistent results, errors, etc.
- Tools
    - pip, Anaconda

ANACONDA®

Managing Projects



Managing Projects



Managing Projects

- **Package managers** allow users to easily install, update, and manage Python packages and libraries
- **Virtual environments** allow users to isolate and manage project-specific dependencies, ensuring that each project has its own clean and independent environment for installing packages and libraries
    - Isolation helps prevent conflicts between different projects and promotes consistency and reproducibility
    - Users can create, activate, deactivate, and delete virtual environments as needed, making it easier to work on multiple projects with different requirements on the same system

Managing Projects

- Images from the scikit-learn documentation
  - If you don't manage your libraries, you'll manually have to update each dependency that does not meet the minimum requirements
- This is why using a **package manager** (pip, Anaconda) and creating new virtual environments (venv, Anaconda) for each project is considered a good practice
  - Anaconda can do both

Installing scikit-learn

Dependency	Minimum Version	Purpose
numpy	1.17.3	build, install
scipy	1.5.0	build, install
joblib	1.1.1	install
threadpoolctl	2.0.0	install
cython	0.29.33	build
matplotlib	3.1.3	benchmark, docs, examples, tests
scikit-image	0.16.2	docs, examples, tests
pandas	1.0.5	benchmark, docs, examples, tests
seaborn	0.9.0	docs, examples
memory_profiler	0.57.0	benchmark, docs
pytest	7.1.2	tests
pytest-cov	2.9.0	tests
ruff	0.0.272	tests
black	23.3.0	tests
mypy	1.3	tests
pyamg	4.0.0	tests
sphinx	6.0.0	docs
sphinx-copybutton	0.5.2	docs
sphinx-gallery	0.10.1	docs
numpydoc	1.2.0	docs, tests
Pillow	7.1.2	docs
pooch	1.6.0	docs, examples, tests
sphinx-prompt	1.3.0	docs
sphinxext-opengraph	0.4.2	docs
plotly	5.14.0	docs, examples
conda-lock	2.0.0	maintenance

Warning: Scikit-learn 0.20 was the last version to support Python 2.7 and Python 3.4. Scikit-learn 0.21 supported Python 3.5-3.7. Scikit-learn 0.22 supported Python 3.5-3.8. Scikit-learn 0.23 - 0.24 require Python 3.6 or newer. Scikit-learn 1.0 supported Python 3.7-3.10. Scikit-learn 1.1 and later requires Python 3.8 or newer.