Python Libraries

CSCI 6118 Software Engineering for Scientists
Jake Krol 2025

Introduction

- Position: Computer science Ph.D. student in Layer lab
- Research: population genomics

2022

Michigan State University

URA

2022 - 2024

University of Colorado Anschutz School of Medicine

RA

2024

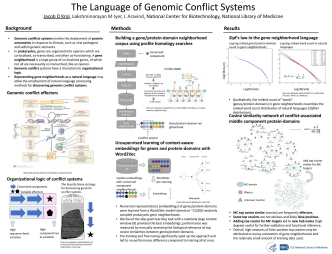
NIH/NLM/NCBI Data Science Internship

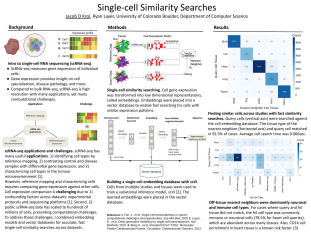
Intern

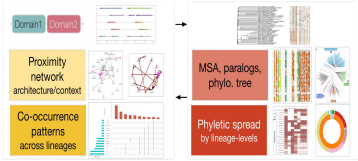
2024

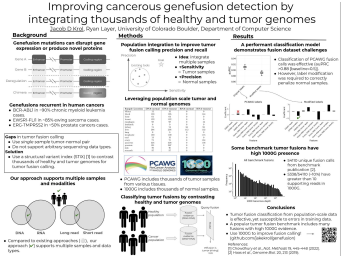
University of Colorado Boulder

Begin Ph.D.





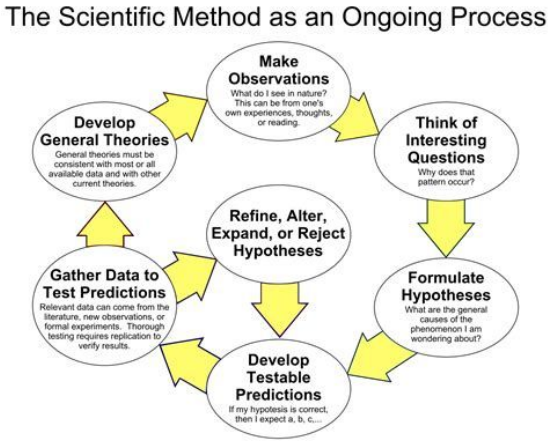




Outline

- Purpose of python libraries
- Terminology
- Brief history
- NumPy
- Pandas
- Matplotlib
- Physical structure of packages

With respect to science



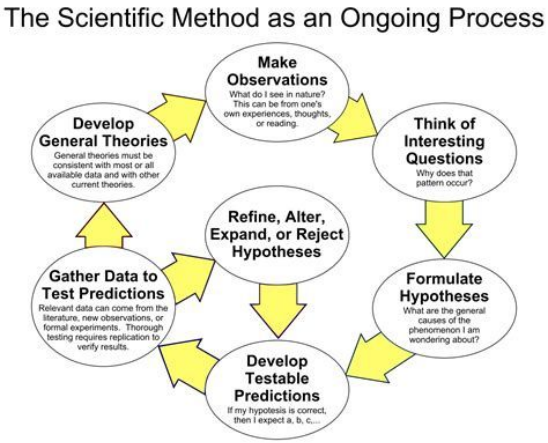
Can we cycle faster using libraries?

Learning goals

- Why are **libraries good for science**?
- What do “**modules**”, “**packages**”, and “**libraries**” mean w.r.t Python?
- How/when use **NumPy**, **Pandas**, **Matplotlib**?
- What alternatives exist for **big data scenarios**?
- What is the **physical structure** of a python package?

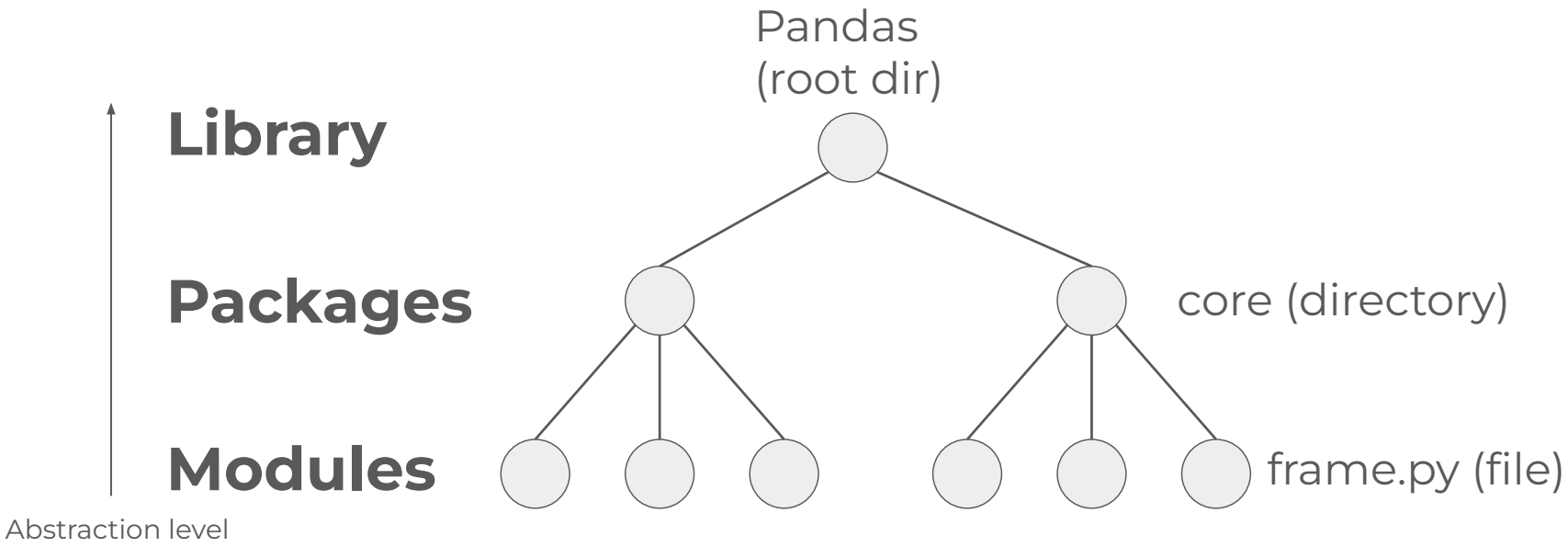
Purpose of libraries

- Formally, a library is a code for general purpose tasks
- $utility_{lib}(trust_{lib}, time, necessity)$
  - $trust_{lib}$
  - time
  - necessity



Terminology

Libraries are composed of **packages** which are composed of **modules**



History of Python packages

- Popular libraries are often “old” and have high *utility*
  - $\text{utility}_{\text{lib}}(\text{trust}_{\text{lib}}, \text{time}, \text{necessity})$

NumPy

- Purpose
  - Init, view, modify, or write multi dimensional arrays
  - Random number generation

NumPy

- Array initializations. Defining and viewing shape

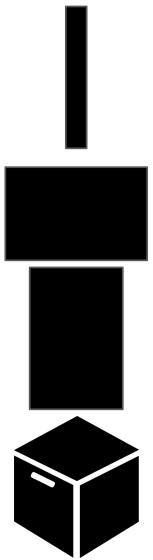
l = list(range(0,12))

a = np.array(l) # vector

np.reshape(a, (3,4)) # matrix

a = np.array(l).reshape(4,3) # matrix

a = np.array(l).reshape(2,3,2) # tensor



Q: What's the difference?

np.reshape(a,(3,4), 'C')
np.reshape(a,(3,4), 'F')

NumPy

- Check shape
  - *np.shape(a)*
- Check # dimensions
  - *np.ndim(a)*
- Q: What is output shape of these two operations?
  - *n = len(a) # a is a vector*
  - ***np.dot**(a,a)*
  - ***np.outer**(a,a)*

?

NumPy

- Elementwise operations

a + 5

a - 5

*a * 10*

a / 2

- NumPy is usually more efficient than base python approaches
  - Low-level C code performs the operations on elements without looping

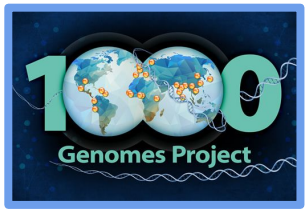
```
a = np.arange(1000000) # 1M

t0 = time.time()
b1 = [i * 2 for i in a]
t1 = time.time()
print("List comprehension:", round(t1 - t0,2), "seconds")
t0 = time.time()
b2 = a * 2
t1 = time.time()
print("NumPy vectorized:", round(t1 - t0,2), "seconds")
```

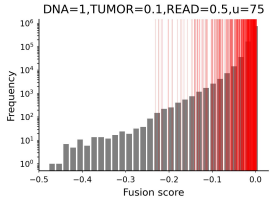
List comprehension: 0.065 seconds
NumPy vectorized: 0.002 seconds

Are 1M elements realistic?

NumPy

- Yes, 1M elements are *common* in scientific applications
- Ex: genomics
  - Say, we inspect 1/10,000th of human genome for variants
    - ~300k base pairs
  - Compare only 10 people
  - We have (3 x 10^5 variants) x (10 people) = 3 x 10^6





My research compares >1000 human genomes

~3 **billion** base pairs in human reference genome

NumPy

- How do non-trivial math quickly on arrays?
  - Numba

```
def f(x):
    if x > 0:
        y = np.log(x)
    else:
        y = x ** 2
```

NumPy v Numba

*Conditional logic must operate on scalars only

Q: how vectorize a user defined function?

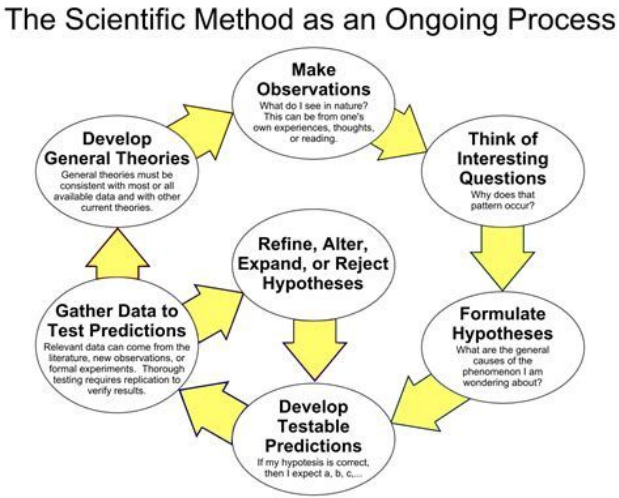
```
x = np.linspace(-1000, 1000, 10_000_000) # data

def piecewise_numpy(x): # original function

    return np.where(x > 0, np.sqrt(x), -np.sqrt(-x))

start = time.time()

y_numpy = piecewise_numpy(x)

numpy_time = time.time() - start

...
```

```
...

@vectorize(["float64(float64)"], target='parallel')
# decorator

def piecewise_numba_vec(x): # numba function

        lambda x: np.sqrt(x) if x > 0 else -np.sqrt(-x)

piecewise_numba_vec(x[:10]) # compile func by
subset run

start = time.time()

y_numba_vec = piecewise_numba_vec(x)

numba_vec_time = time.time() - start
```

Original: 0.09s
Numba: 0.01s

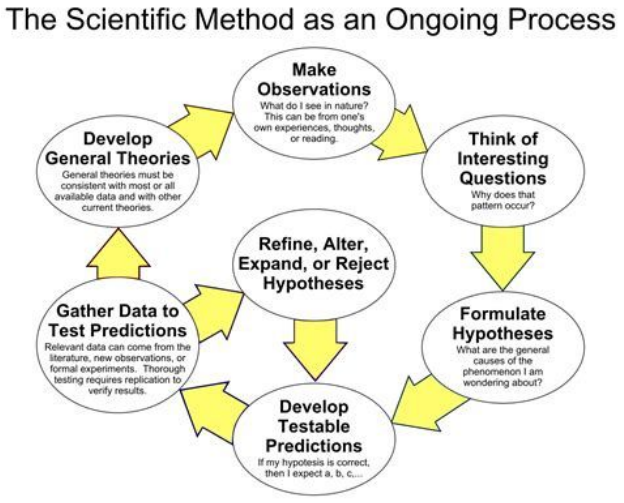
Python v NumPy v Numba

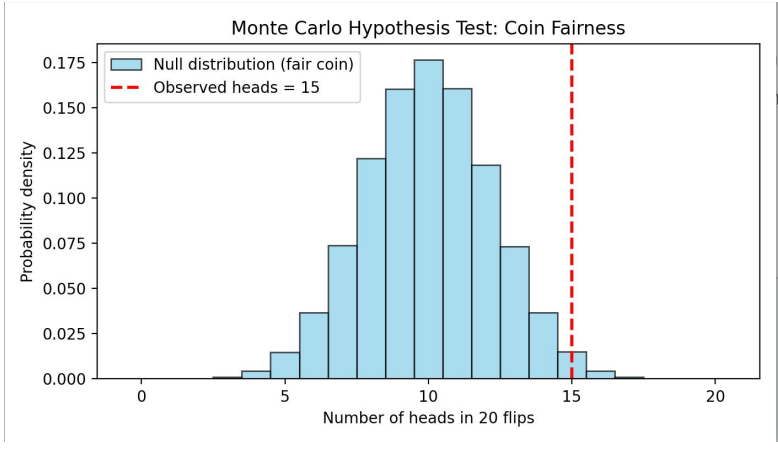
- Python takes ~x100 longer than Numba
- NumPy takes ~x10 longer than Numba



NumPy

- Random number generation
- Applications
  - Monte carlo hypothesis testing
  - Simulations
- Simulations are a cheap, fast way to test experimental setups, hypotheses, etc.
  - **Expected data can test expected results**
  - Proactively setup pipelines before real data is collected





NumPy

n_flips = 20

heads_emp = 15

n_permutations = 10000

simulate 20 fair coins flips 10k times

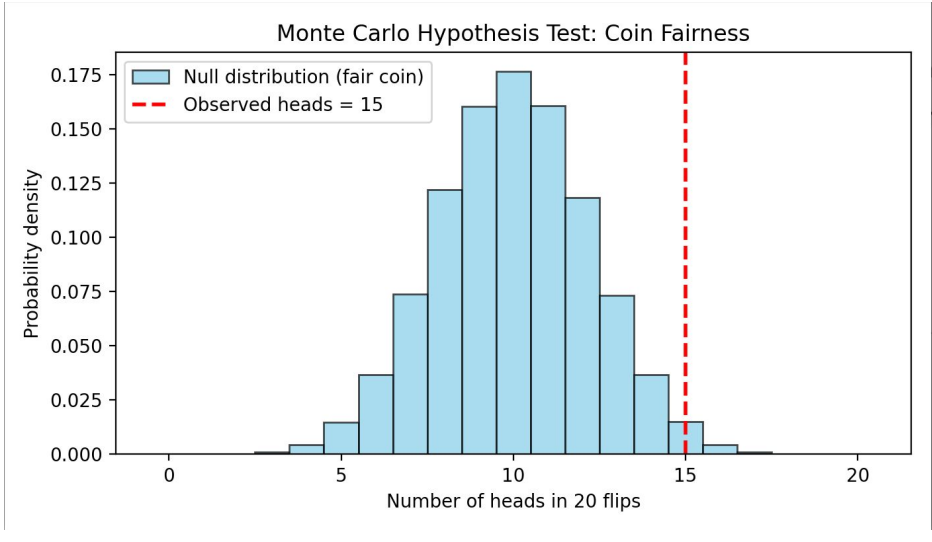
simulated_heads = np.random.binomial(n_flips, 0.5, n_permutations)

compute one-sided p-value (alternative: more heads than expected)

num_null_extreme = np.sum(simulated_heads >= empirical_heads)

p_value = (num_null_extreme + 1) / (n_permutations + 1) # +1 for empirical

One-sided p-value: 0.0225



Numpy

- Support for various distribution functions

?np.random ->

Univariate
distributions

---
beta Beta distribution over ``[0, 1]``.
binomial Binomial distribution.
chisquare :math:`\chi^2` distribution.
exponential Exponential distribution.
f F (Fisher-Snedecor) distribution.
gamma Gamma distribution.
geometric Geometric distribution.
gumbel Gumbel distribution.
hypergeometric Hypergeometric distribution.
laplace Laplace distribution.
logistic Logistic distribution.
lognormal Log-normal distribution.
logseries Logarithmic series distribution.
negative_binomial Negative binomial
distribution.
noncentral_chisquare Non-central chi-square
distribution.
noncentral_f Non-central F distribution.
normal Normal / Gaussian distribution.
pareto Pareto distribution.
poisson Poisson distribution.
power Power distribution.
rayleigh Rayleigh distribution.
triangular Triangular distribution.
uniform Uniform distribution.
vonmises Von Mises circular distribution.
wald Wald (inverse Gaussian) distribution.
weibull Weibull distribution.
zipf Zipf's distribution over ranked data.

NumPy

- Summary
  - For “box” data
  - Fast numeric computation
  - Simulation from various distributions
- Advanced
  - Vectorized functions for large datasets (numba or built-ins)
  - Specialized compression formats for faster I/O
    - Look into *np.savez(file, arrays)*
- Limitations
  - Best for *numeric* data
  - Pandas is usually better for tables with mixed data types (strings + numbers)

Pandas

- Purpose
  - Read, init, view, modify, or write tabular data
- How Pandas differ from NumPy?
  - **Optimal for 2D** data only
  - Better at “**nuancing**” which values undergo an **operation**
    - e.g., *groupby()*
- When applicable, Pandas will use NumPy for numeric operations

Pandas

- Basics

*df = pd.DataFrame({"x": [1, 2,], "y": [3,4]}) # **init***

*df.to_csv("example.tsv", index=False, sep='\t') # **write***

*df = pd.read_csv("example.tsv", sep='\t') # **read***

*df['z'] = df.apply(axis=1, func = lambda row: row['x'] + row['y']) # **row-wise operation***

*df['u'] = df['x'].apply(func=lambda i: i**2) # **column-wise operation***

- So far, similar to NumPy.

Pandas

- LOCating values

*df = pd.DataFrame({"x": [1, 2,], "y": [3,4]}) # **init***

*df.iloc[1,0] # **numeric indexing** (0-based)*

Q: which value is returned?

*df.loc[0,'x'] # **name indexing** (by default, row indices are already integers)*

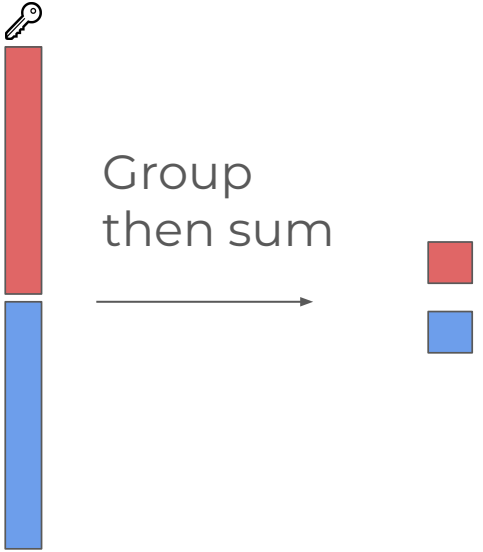
df['x'] # by default, name indexing searches the columns for matching

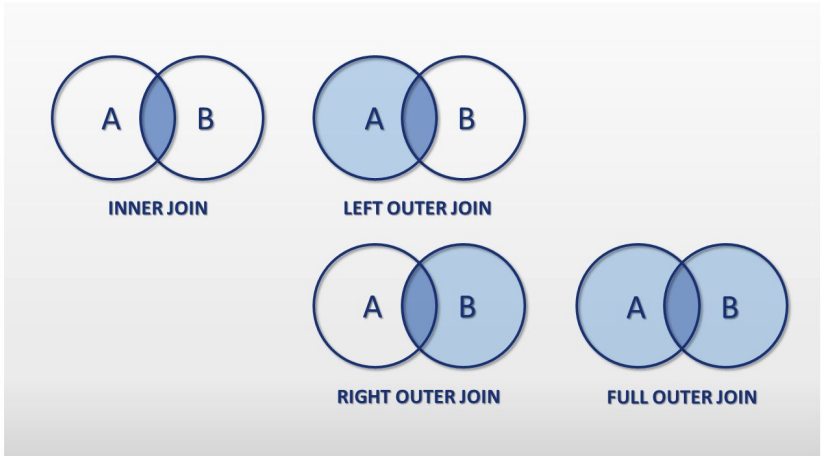
*df.columns # **column name extraction***

Pandas

- Common uses
  - **Key**-dependent operations
    - Groupby(<**key**>)
    - pd.merge(x,y,on=<**key**>)
    - <key> is a column name
  - Again, Pandas is great for “nuancing” which data subsets operations apply to

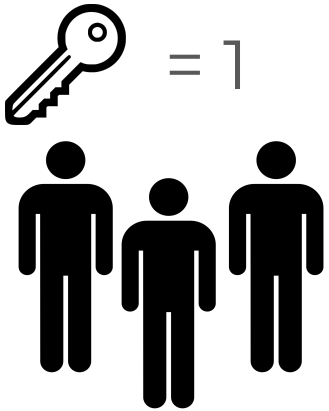
= ?

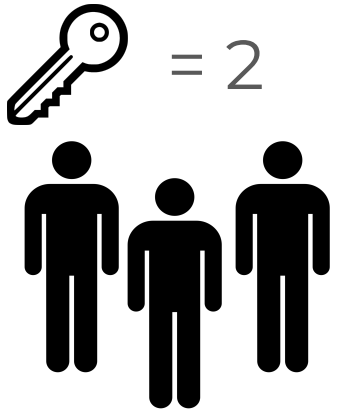




Pandas

- In the context of Pandas, **keys** are tags that categorize items (usually rows).





Pandas

- Groupby
  - 1. Collect all rows with shared key
  - 2. Perform a *summary* operation on groups

df = pd.DataFrame({'key': ['a','a','b','b'], 'val': [-1,1,2,4]})

grouped =df.groupby('key')

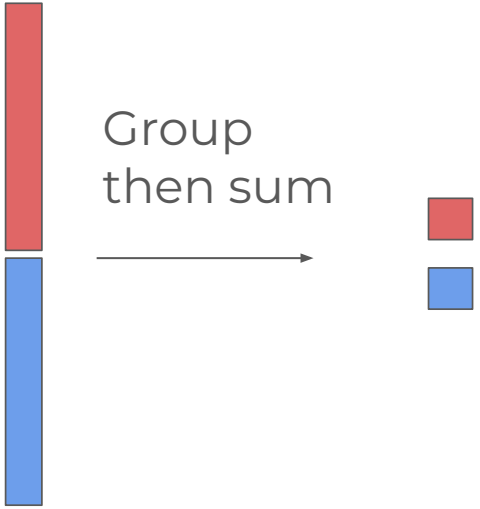
grouped.sum()

 # outputs a len(unique_keys) x 2 table

 # key value

 # a 0

 # b 6



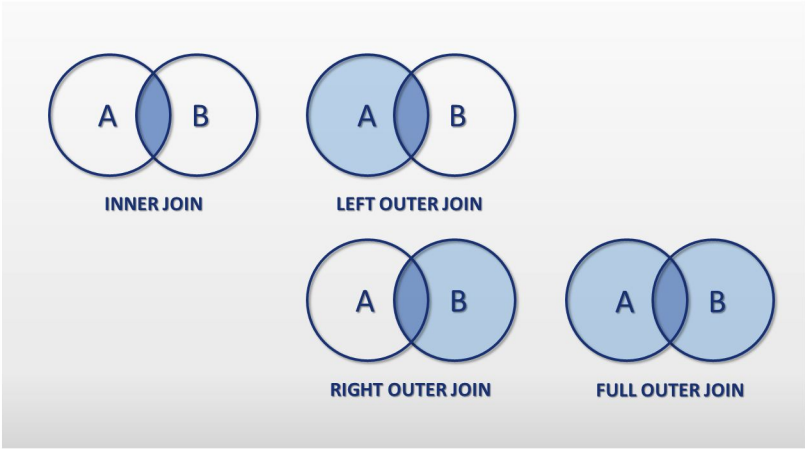
What other group summary funcs are built-in?

print(';'.join(dir(grouped))

agg;aggregate;all;any;apply;bfill;boxplot;corr;corrwith;count;cov;cumcount;cummax;cummin;cumprod;cumsum;describe;diff;dtypes;ewm;expanding;ffill;fillna;filter;first;get_group;groups;head;hist;idxmax;idxmin;indices;key;last;**max;mean;median;min**;ndim;ngroup;ngroups;nth;nunique;ohlc;pct_change;pipe;plot;prod;quantile;rank;resample;rolling;sample;sem;shift;size;skew;std;sum;tail;take;transform;value_counts;var

Pandas

- Join
  - Combine >=2 datasets by a shared key
  - Types
    - Inner: intersection
    - Left: A and intersection
    - Right: B and intersection
    - Outer: All keys



Pandas

```
A = pd.DataFrame({
        "key": ["a", "a", "b", "b"],
        "val_a": [-1, 1, 2, 4]
})
B = pd.DataFrame({
        "key": ["a", "b", "c"],
        "val_b": ['x', 'y', 'z']
})
pd.merge(A,B,on='key',how='left')
```

Q: What shape will the output be?

Pandas

```
A = pd.DataFrame({
        "key": ["a", "a", "b", "b"],
        "val_a": [-1, 1, 2, 4]
})
B = pd.DataFrame({
        "key": ["a", "b", "c"],
        "val_b": ['x', 'y', 'z']
})
pd.merge(A,B,on='key',how='left')
```

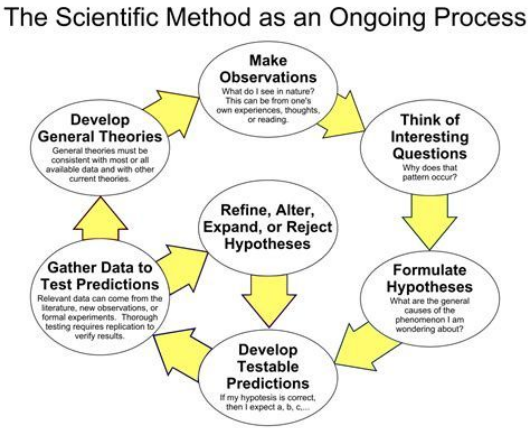
```
# 4 x 3
key  val_a val_b
 a     -1     x
 a      1     x
 b      2     y
 b      4     y
```

Q: what shape would an outer join be?

A: 5 x 3

Pandas

- Limitation
  - Row-wise operations can be very slow
    - Solution: swifter
  - I/O is slow for large data and memory usage is high
    - Alternative: duckdb



Pandas

- Use **swifter** library for **really easy rowwise parallelization**

```
# used 50 cores for experiment
N = 10_000_000
df = pd.DataFrame({
        a': np.random.rand(N),
        'b': np.random.rand(N),
        'c': np.random.rand(N)
})
def rowf(row):
        return math.exp(row.a) \
                * math.sin(row.b) \
                + math.log1p(row.c**2)
```

```
result_pandas = df.apply(rowf, axis=1)
result_swifter = df.swifter.apply(rowf,
axis=1)
```

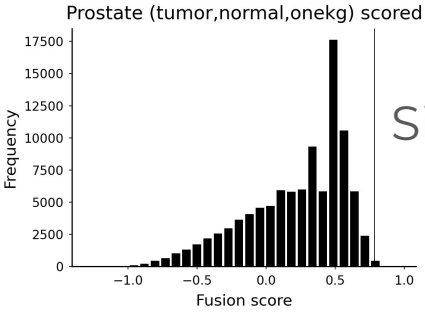
Pandas: 97.39 s
Swifter: 16.27 s

Pandas

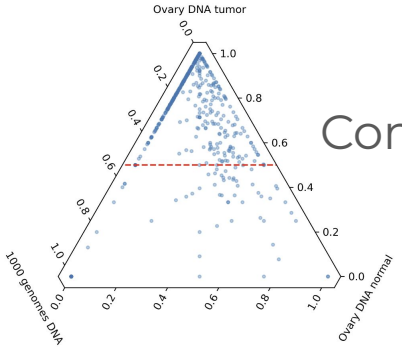
- Summary
  - 2D data
  - Better non-numeric support than NumPy
    - Keys and indexing enable ...
      - “Nuanced” operation scope
      - Relate >=2 datasets
- Pandas has some built-in plotting support
  - Ex: df[‘x’].hist()
- However, matplotlib code is required to control format

Matplotlib

- Purpose
  - Data visualization
- For science communication,
  - Plots should be easy to interpret
  - Science is complex, and the goal of communication is to convey key takeaways to our audience



Simple

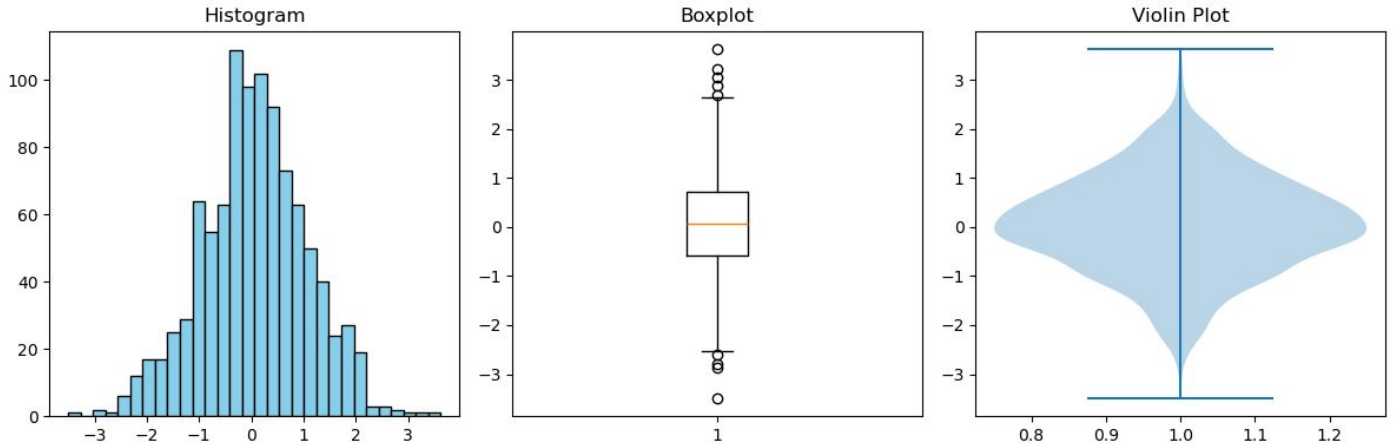


Complex

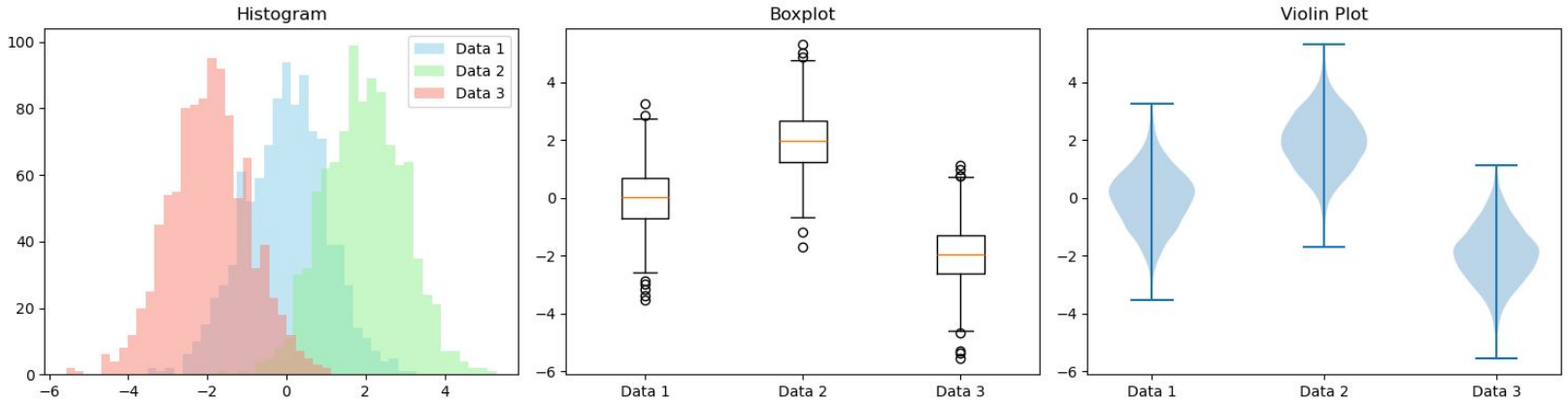
Matplotlib

- 1-dimensional input data
  - Histogram
  - Boxplot
  - Violin

Q: Which should you choose?



Matplotlib



Use cases

- Distribution analysis for ~<=3 empirical vectors
- Median and IQR comparison for many empirical vectors
- Distribution analysis for ~>3 emprical vectors
- My least fav.

Matplotlib

- 1-dimensional plotting code

import matplotlib.pyplot as plt; import numpy as np

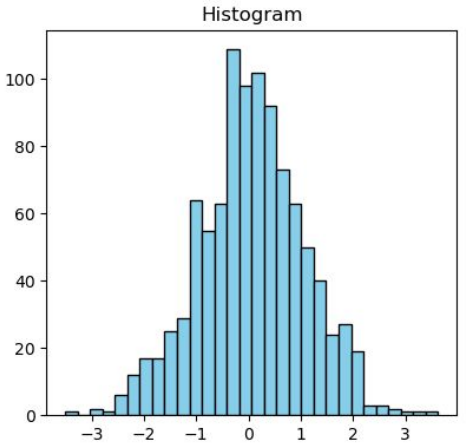
data = np.random.randn(1000)

plt.hist(data,bins=30,color='skyblue',edgecolor='black')

plt.title('Histogram')

plt.show() # for interactive sessions

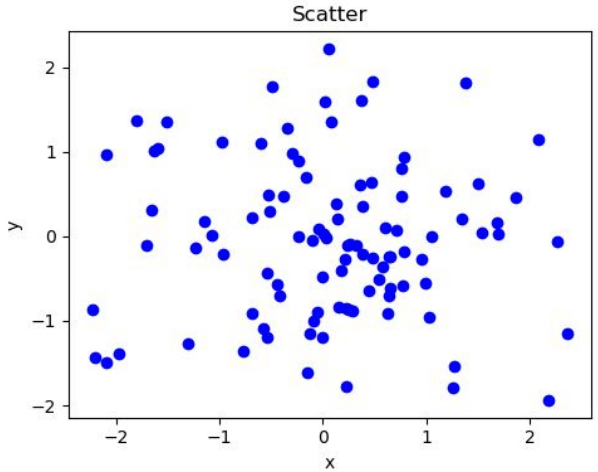
plt.savefig('hist.png') # save to png file

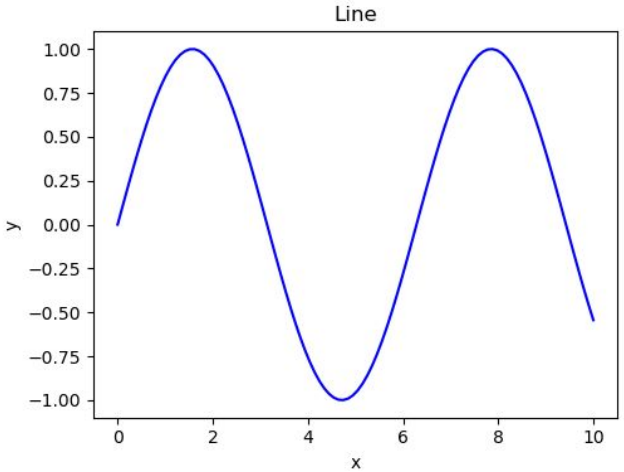


Matplotlib

- 2-dimensional input data
  - Scatter
  - Line

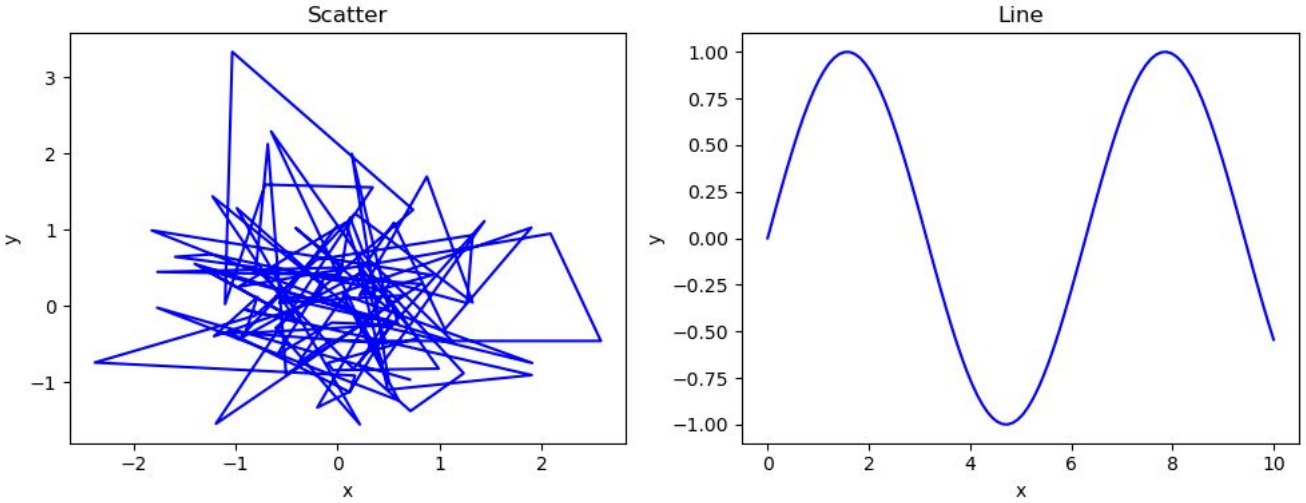
Q: what would line plot of left look like?



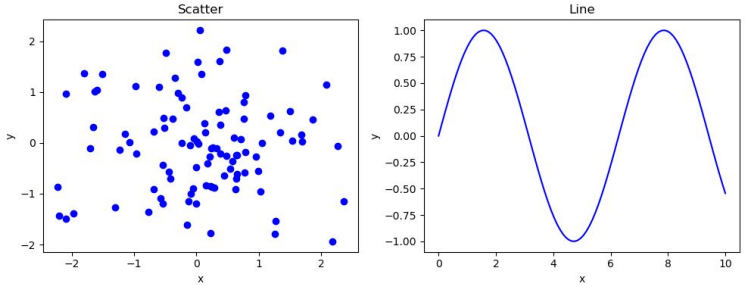


Matplotlib

- **Line** plots make sense for **one-to-one** mappings (proper functions)
- **Scatter** plots are good for **empirical data**



Matplotlib



- Scatter and line plotting code

scatter

 # standard normal sampling

x = np.random.randn(100)

y = np.random.randn(100)

plt.scatter(x,y)

plt.show()

line

 # 100 values eq spaced from [0,10]

x = np.linspace(0,10,100)

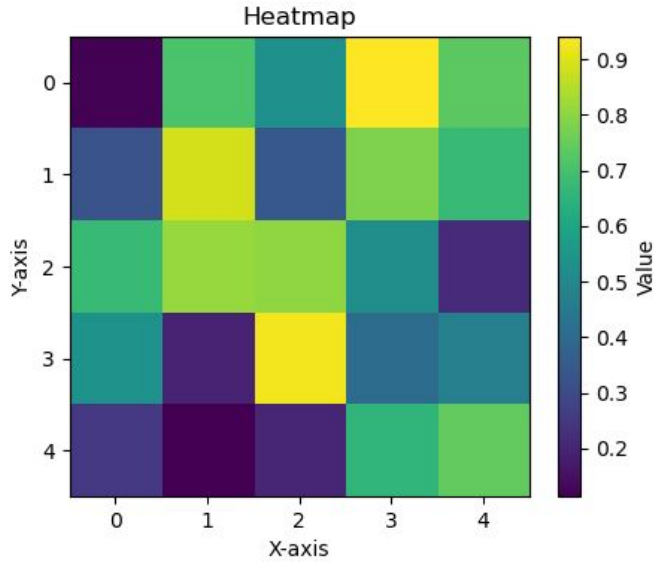
y = np.sin(x)

plt.plot(x,y)

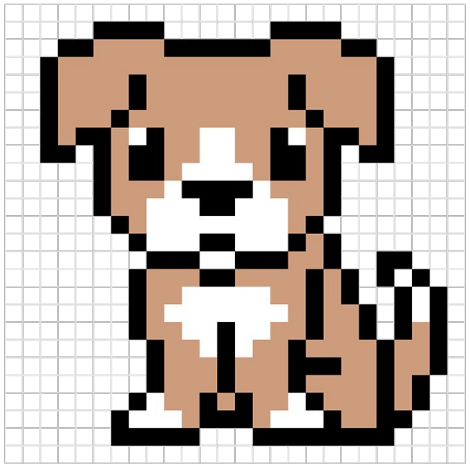
plt.show()

Matplotlib

- 3-dimensional input data
  - Heatmap



=



Matplotlib

- Heatmap plotting code

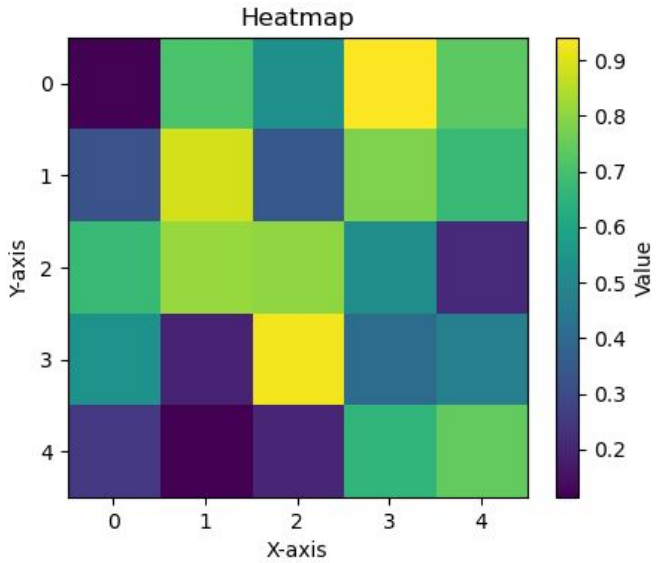
 # 5x5 array sampled from [0,1) unif dist

data = np.random.rand(5,5)

 # heatmaps are equivalently images

plt.imshow(data,cmap='viridis')

plt.colorbar(label='Value')



Matplotlib

- Title, axes, and formatting

plt.title()

plt.xlabel()

plt.ylabel()

 # set size (inches) upon init

plt.figure(figsize=(8,6))

 # get axes object of plot

ax = plt.gca()

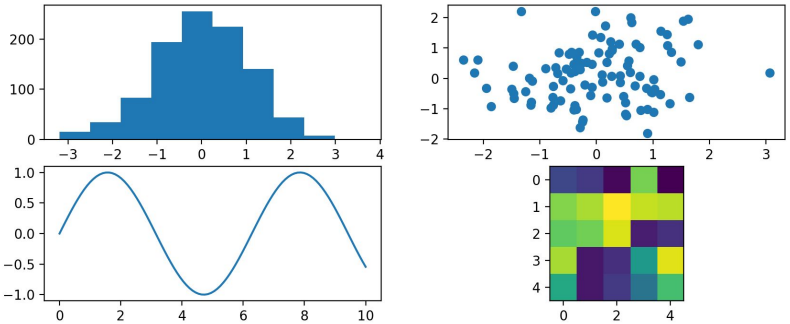
 # rm top boundary line

ax.spines['top'].set_visible(False)

Matplotlib

- Object oriented approach has finer control, but diff syntax
- Allows for subplots

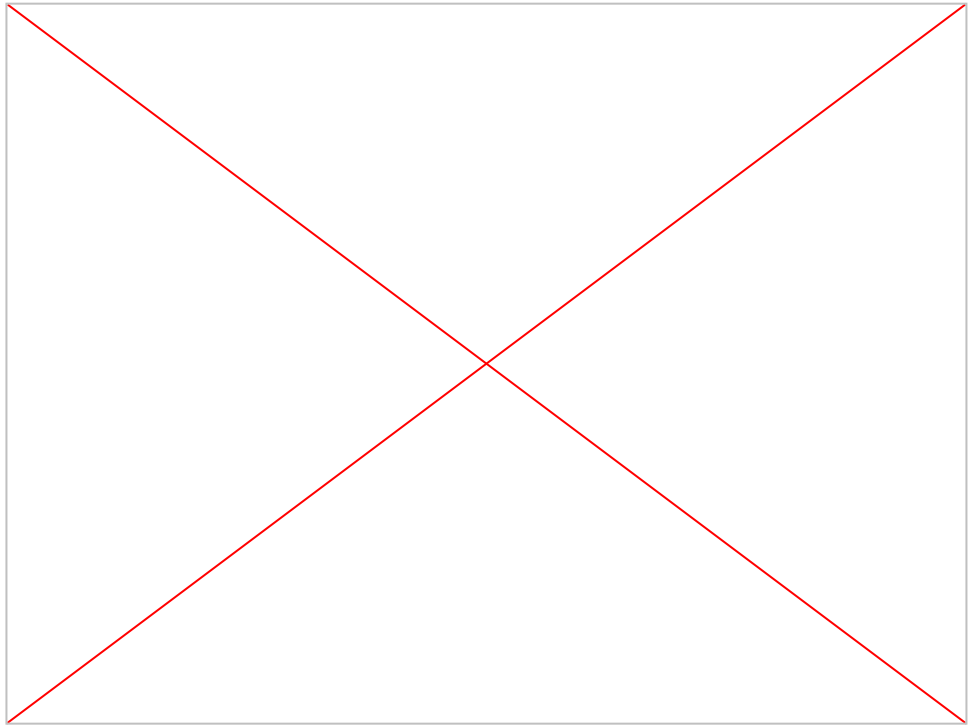
fig, axes = plt.subplots(2, 2, figsize=(10, 4))
axes[0,0].hist(np.random.randn(1000))
axes[0,1].scatter(np.random.randn(100),np.random.randn(100))
axes[1,0].plot(np.linspace(0,10,100), np.sin(np.linspace(0,10,100))
axes[1,1].imshow(np.random.rand(5,5), cmap='viridis')

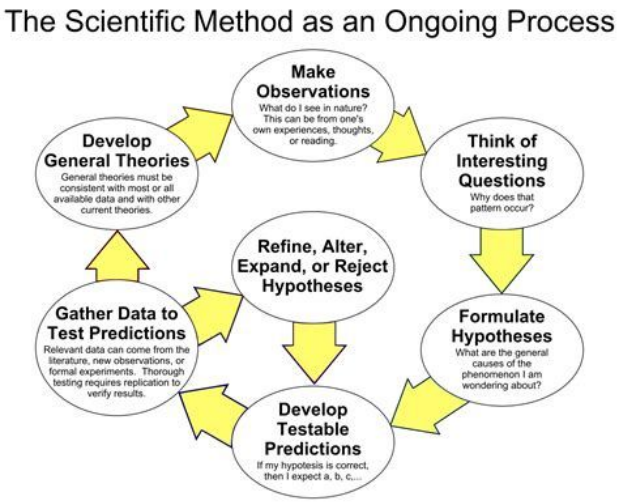


Matplotlib v Plotly

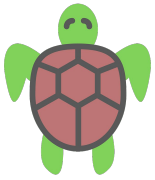
- Limitation
  - Static outputs
- Alternative plotly
  - *__Caution:__ interactive plots are *good for doing* science, but *not great for communicating* science
    - Do you expect a reviewer to zoom/pan to the correct location?
      - They may not even open the file
    - Consider using gifs/videos if using interactive plots in presentations

Matplotlib v Plotly

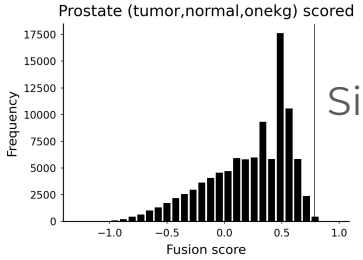




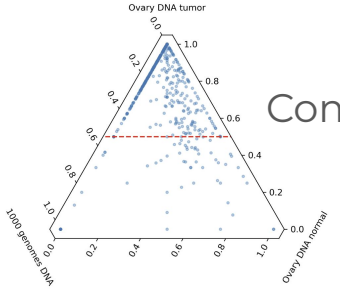
$\text{utility}_{\text{lib}}(\text{trust}_{\text{lib}}, \textbf{time}, \textbf{necessity})$



Matplotlib

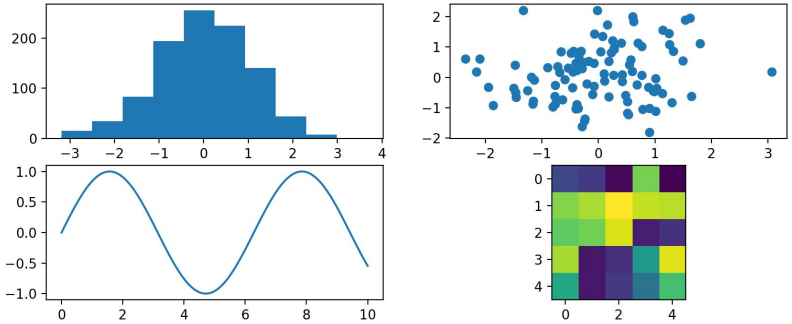


Simple



Complex

- Summary
  - The goto python visualization library
  - Examples for 1/2/3-dimensional input data
  - Best for static plots
    - Plotly is an interactive alternative
- Remember
  - Plots should communicate science efficiently; beware complex plots



Beyond NumPy Pandas Matplotlib

- Scipy: stats and optimization
- Scikit-learn: machine learning
- Pytorch: deep learning
- Networkx: network analysis
- Pyvis: interactive network visualization

Which libraries will emerge next?

Physical structure of package

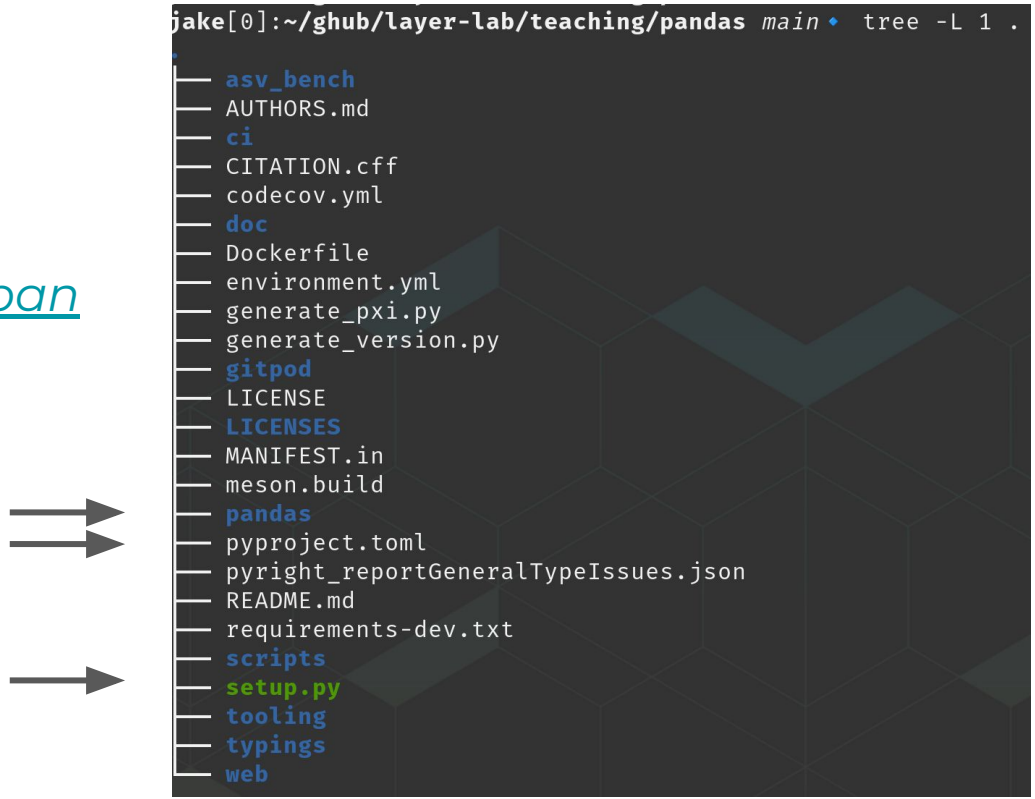
- Scenario: You need to write/edit a python package for your science project
- Approach
  - 1. Install an editable version
  - 2. Implement changes
  - 3. Deploy
- Free deployment at https://pypi.org/

Physical structure of package

- Editing pandas library

git clone

https://github.com/pandas-dev/pandas.git



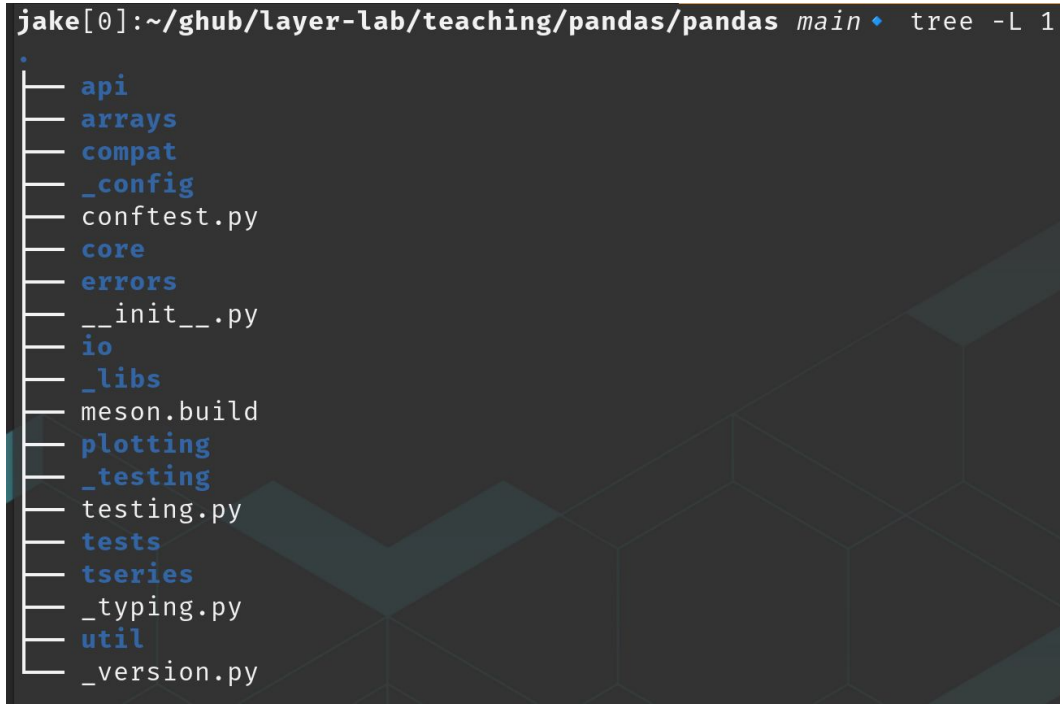
Physical structure of package

- **pyproject.toml** OR **setup.py** have pkg metadata

```
[project]
name = 'pandas'
dynamic = [
  'version'
]
description = 'Powerful data structures for data analysis, time series, and statist
ics'
readme = 'README.md'
authors = [
  { name = 'The Pandas Development Team', email='pandas-dev@python.org' },
]
license = {file = 'LICENSE'}
requires-python = '>=3.11'
dependencies = [
  "numpy>=1.26.0",
  "python-dateutil>=2.8.2",
  "tzdata>=2023.3"
```

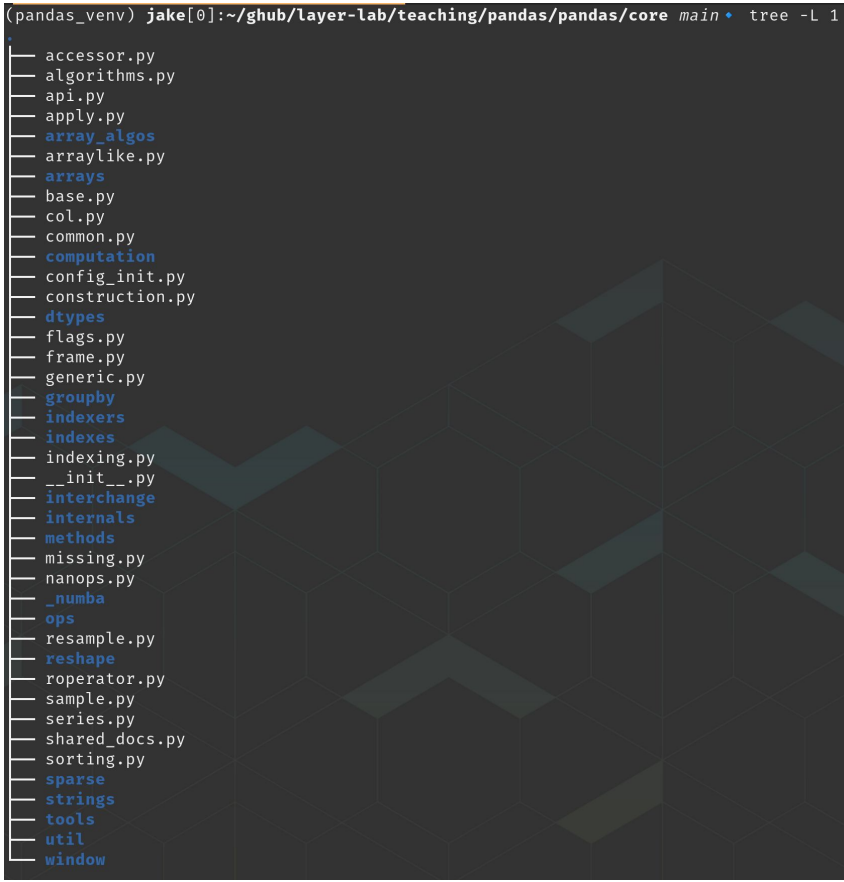
Physical structure of package

- Core
  - Key functions and classes
- __init__.py
  - Declares as package



Physical structure of package

- What's inside core package?
  - frame.py, series.py, ...



Physical structure of package

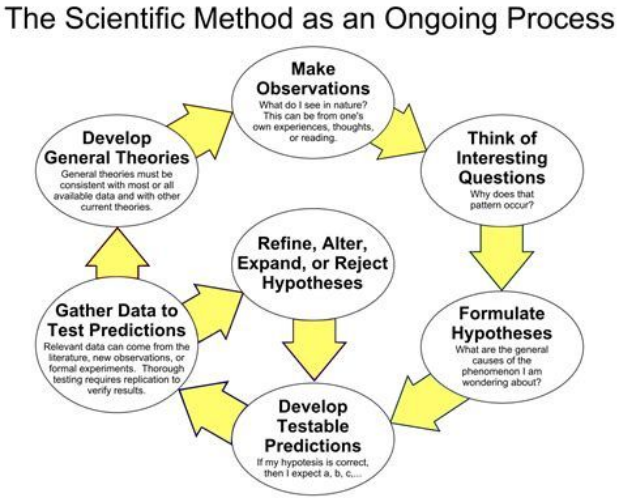
- Adding a print statement to shape attribute

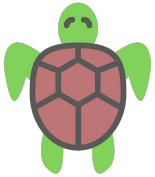
```python
@property
def shape(self) -> tuple[int, int]:
    """
    Return a tuple representing the dimensionality of the DataFram

    Unlike the `len()` method, which only returns the number of ro
    provides both row and column counts, making it a more informat
    understanding dataset size.

    See Also
    --------
    numpy.ndarray.shape : Tuple of array dimensions.

    Examples
    --------
    >>> df = pd.DataFrame({"col1": [1, 2], "col2": [3, 4]})
    >>> df.shape
    (2, 2)

    >>> df = pd.DataFrame({"col1": [1, 2], "col2": [3, 4], "col3":
    >>> df.shape
    (2, 3)
    """
    print('hello from shape method')
    return len(self.index), len(self.columns)
```

Physical structure of package

- When write your own python package?
- Opinion: only if you *absolutely have to*
- A very simple pkg example
  - https://github.com/jakekrol/aibou

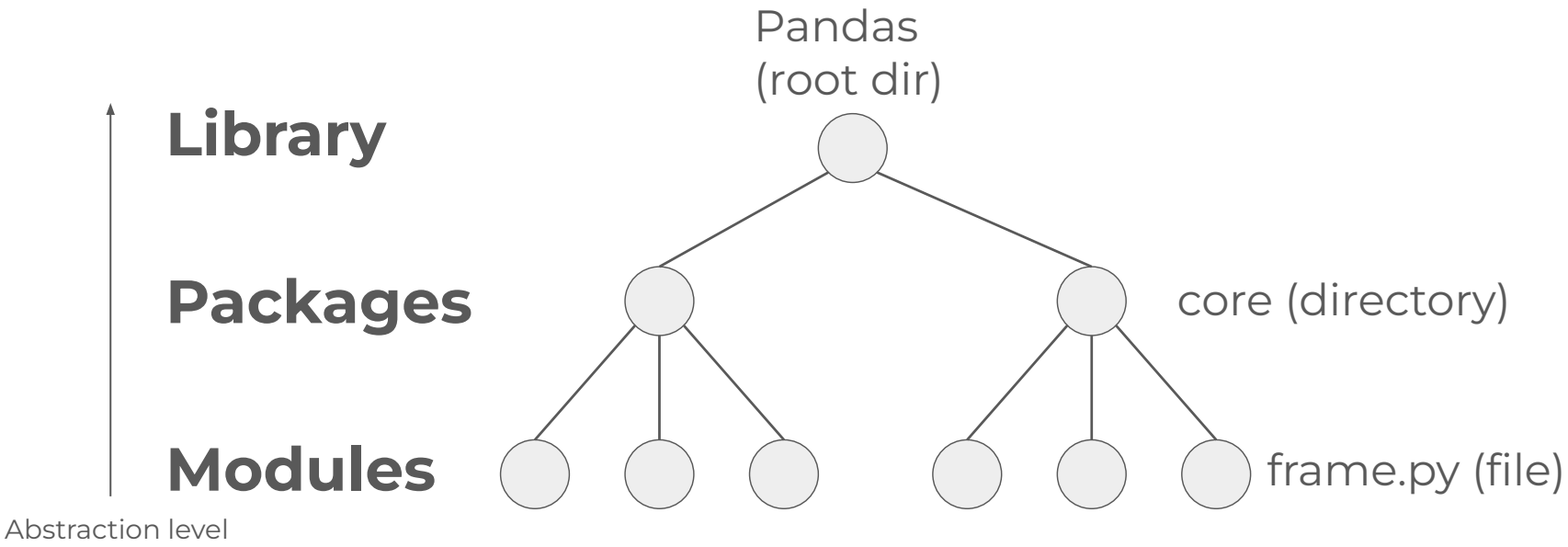
aibou	hotfix: add missing __init__.py	2 years ago
screenshots	screenshots of game	2 years ago
.gitignore	pypi-release: aibou 1.0.6	2 years ago
LICENSE	pypi-release: aibou 1.0.6	2 years ago
MANIFEST.in	pypi-release: aibou 1.0.6	2 years ago
README.md	README: grammar	2 years ago
pyproject.toml	pypi-release: aibou 1.0.6	2 years ago





Physical structure of package

- Pandas is just a directory with python code and package metadata



Conclusions

- Python libraries can speed up science
- **NumPy**: numeric processing & random number generation
- **Pandas**: “nuanced” tabular operations
- **Matplotlib**: visualization
- Limitations and alternatives for each
- “**Libraries**” (dir) are composed of “**packages**” (dir) which are composed of “**modules**” (files)