NumPy Arrays

1405 Instructor: Ruiqing (Sam) Cao

Required Python Libraries for Today

Core

- NumPy, Pandas

```
import numpy as np
import pandas as pd
```

Visualization

- Matplotlib, Seaborn

```
from matplotlib import pyplot as plt
import seaborn as sns
```

Statistical learning

- scikit-learn, SciPy, statsmodels

```
import sklearn
import scipy
import statsmodel as sm
```

NumPy & Pandas Printing Format

It's better to print only *the first few decimal digits* of large real numbers. Set the print options to keep 3 decimal digits and supress scientific notation (for NumPy arrays and Pandas DataFrames):

- **NumPy**

```
np.set_printoptions(precision=3,suppress=True)
```

- **Pandas**

```
pd.options.display.float_format = '{:.3f}'.format
```

NumPy Array

- A NumPy array is a very useful data structure for storing *large, multi-dimensional, homogenous* data for efficient *numerical computations* (requires importing the `numpy` library)

Key features:

- ➢Homogeneity: (1) values are of the same scalar type and often numeric (2) rectangular so all elements have the exact same size
- ➢Multidimensional: 1D, 2D, or higher-dimension (1D & 1D most common)
- ➢Efficiency: faster than lists for numeric computations, and supports built-in mathematic and statistics functions

NumPy Array Examples

`['abc',12]` not a NumPy array → `'abc'` and `12` are different data types

`[[1,2],[3]]` not a NumPy array → `[1,2]` and `[3]` have different sizes

`[[[1,2],[3,4]],[[5,6],[7,8]]]` is a NumPy array of shape (2, 2, 2)

`[0.1, 0.5, 1.0, 1.5]` is a NumPy array of shape (4,) of float type

2D Array As a Sequence of 1D Arrays

- We can think of a 2D array `arr` as either
  - ➢A sequence of 1D arrays as rows (of the same length), or
  - ➢A sequence of 1D arrays as columns (of the same length)
- For `arr[i,j]`, some important concepts and their relationships
  - `i` is a *row* index: `axis=0`, and number of rows equals `arr.shape[0]`
  - `j` is a *column* index: `axis=1` and number of columns equals `arr.shape[1]`

	Rows (axis=0)	**Columns (axis=1)**
Role	Represents an Observation	Represents a Variable
Access	`array[i,:]` (row `i`)	`array[:,j]` (column `j`)
Size	`array.shape[0]`	`array.shape[1]`

Common Ways to Initialize an Array

- Directly rom a list: `np.array([0.1,0.2,0.3,0.4],dtype=np.float64)`
- Zeros: `np.zeros(5, dtype=np.float64)` → array([0., 0., 0., 0., 0.])
- Ones: `np.ones(5, dtype=np.float64)` → array([1., 1., 1., 1., 1.])
- Uninitialized values (empties)

`np.empty([2,2], dtype=np.int64)` → An arbitrary 2x2 ndarray

- Sequence with specified *start*, *stop*, & *step*

`np.arange(10)` → array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

`np.arange(0.1, 1.05, 0.05)` → array([0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5])

Common NumPy Data Types (Numeric)

Data type	Type description	Range (int/bool) or Precision (float)
np.int8	8-bit integer	-128 to 127
np.int16	16-bit integer	-32768 to 32767
np.int32	32-bit integer	-2^31 to 2^31-1 (~9 digits)
np.int64	64-bit integer	-2^63 to 2^63-1 (~19 digits)
np.float16	Half-precision floating point	3-4 decimal digits
np.float32	Single-precision floating point	7 decimal digits
np.float64	double-precision floating point	15-16 decimal digits
np.float128	Extended-precision floating point	33 decimal digits
np.bool_	Boolean type	{True, False}
np.string_	Fixed-length ASCII string type	

Note trailing underscores!

NumPy Data Type Casting

- Automatically conversion of array elements of different types: bool_ → int* → float* → string_

For example:

`np.array([1,'two',3.0])` → array(['1', 'two', '3.0'], dtype='<U32')

Unicode string of length 32

`np.array([100,10.0,False])` → np.array([1,'two',3.0])

NumPy Data Type Casting

- Explicit type casting, two approaches `arr=np.array([0, 1, 2, 3, 4])`:
  - `arr.astype(np.float64)`
  - `np.float(arr)`

Cast the array into float type and give the same result

→ array([0., 1., 2., 3., 4.])

- Caution: casting from coarse to precise is okay (e.g., int to float), but try to avoid casting from precise to coarse, for example
  - Allowed but loses precision: `np.int64(100.9)`
  - Allowed but messing up completely: `np.int16(1000000)`
  - Not allowed and returns an error: `np.float32('hello world')`

Keeps integer part only

Out of range

Representing Missing Values

A numeric missing value is represented by `np.nan` (“not a number”)

- Note: This does not apply to strings, where missing values are simply denoted by NoneType (None) or empty string ("")

- *Comparison logics don’t work* on the missing value
  - Any comparison between a non-missing value `x` and `np.nan` results in `False`: e.g., `np.nan<0` and `np.nan>=0` both evaluate to `False`

- In general, final data *should not contain* `np.nan`; clean the data to remove or fill missing values before proceeding with data analysis

Exercise: Create Arrays & Cast Types

1. Create a NumPy array of integers from 0 to 999 with `dtype= np.int16`, then convert it to `dtype= np.float32`

2. Create a NumPy array of 1000 zeros with `dtype=np.float16`, then convert it to `dtype=np.bool`

3. Generate a 2D NumPy array with 100 rows and 3 columns:
   - *Column 0*: row numbers 0-99
   - *Column 1*: Continuous values drawn from a uniform distribution between 0 and 1
   - *Column 2*: Binary values (0 or 1) drawn from an uneven coin flip where heads (1) has a probability of 0.6, and tails (0) has a probability of 0.4

➢ *Hints: see `np.random.rand()` and `np.random.choice()` examples in Notebook*

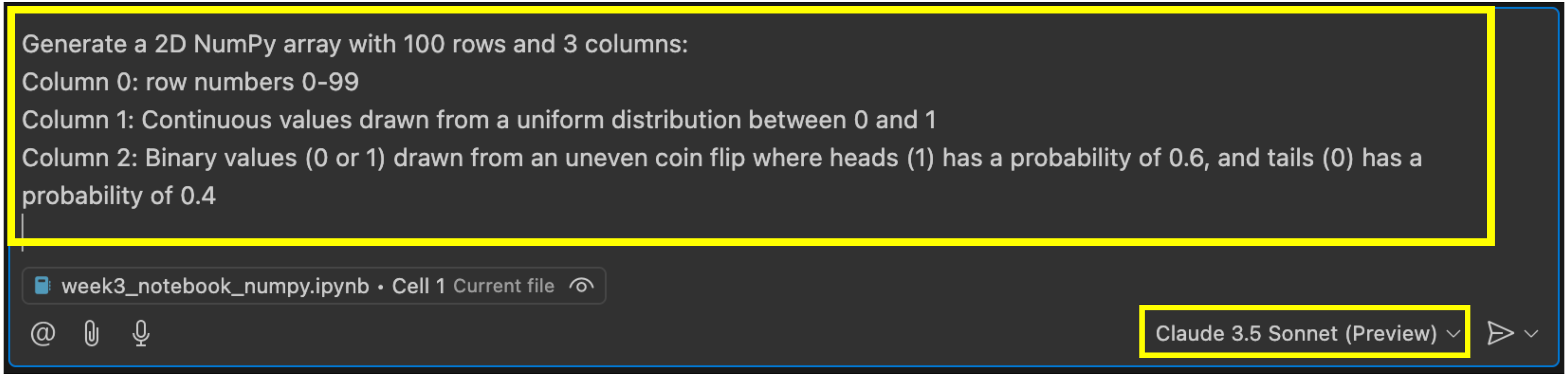
Using Generate AI to Generate Code

Prompt

- Same text as the page's problem instructions. **Clear and step-by-step instructions** are often enough to get Copilot to generate good code

Model

- The model used to generate the code here is **Claude 3.5 Sonnet**



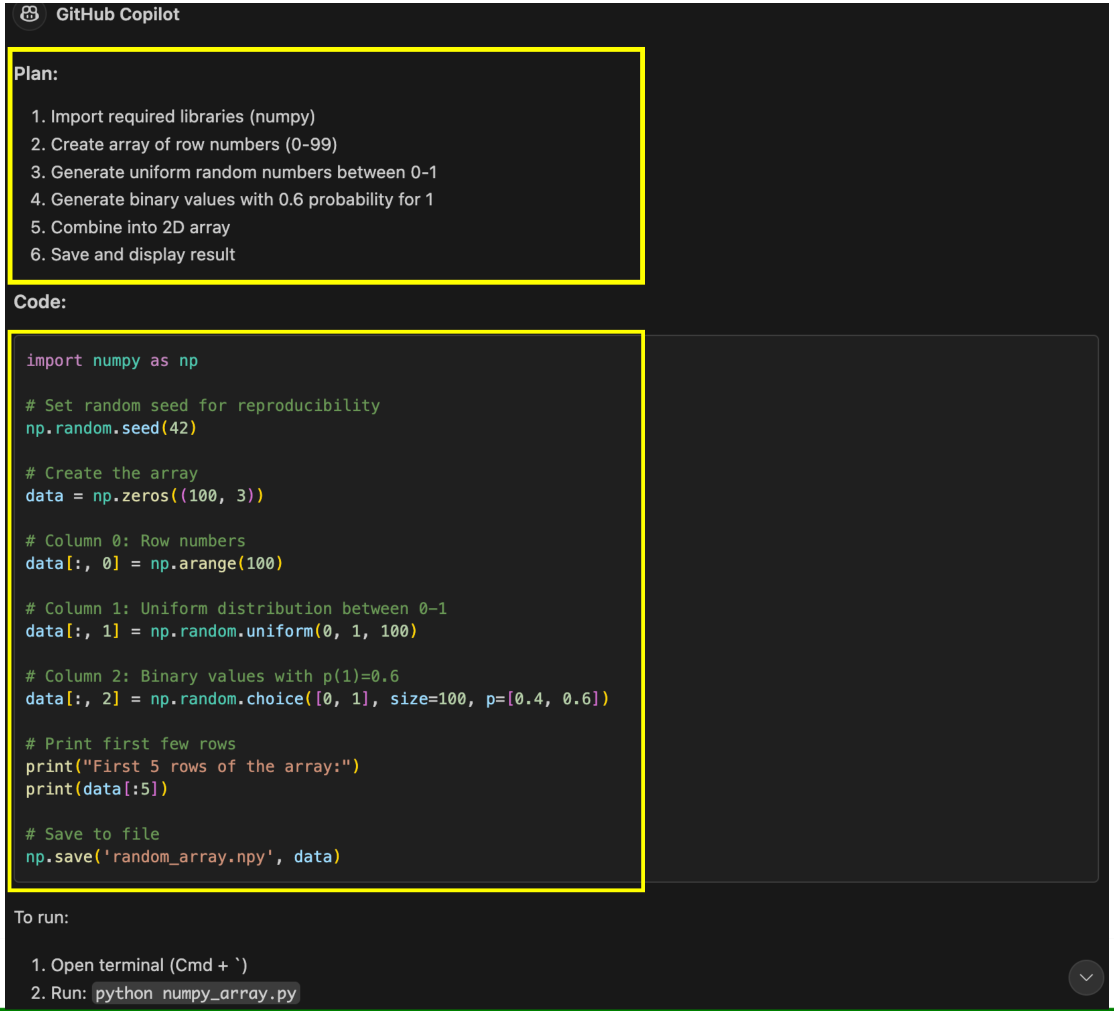
Generate a 2D NumPy array with 100 rows and 3 columns:
Column 0: row numbers 0-99
Column 1: Continuous values drawn from a uniform distribution between 0 and 1
Column 2: Binary values (0 or 1) drawn from an uneven coin flip where heads (1) has a probability of 0.6, and tails (0) has a probability of 0.4

week3_notebook_numpy.ipynb • Cell 1 Current file
@
Claude 3.5 Sonnet (Preview)

Using Generate AI to Generate Code

Copilot Output

- Starts with a description or **<u>plan</u>** for generating the code
- Then provides the Python **<u>code</u>**
- *Note: you may get a different response or code, but they should work*



Plan:

1. Import required libraries (numpy)
2. Create array of row numbers (0-99)
3. Generate uniform random numbers between 0-1
4. Generate binary values with 0.6 probability for 1
5. Combine into 2D array
6. Save and display result

Code:

```
import numpy as np

# Set random seed for reproducibility
np.random.seed(42)

# Create the array
data = np.zeros((100, 3))

# Column 0: Row numbers
data[:, 0] = np.arange(100)

# Column 1: Uniform distribution between 0-1
data[:, 1] = np.random.uniform(0, 1, 100)

# Column 2: Binary values with p(1)=0.6
data[:, 2] = np.random.choice([0, 1], size=100, p=[0.4, 0.6])

# Print first few rows
print("First 5 rows of the array:")
print(data[:5])

# Save to file
np.save('random_array.npy', data)
```

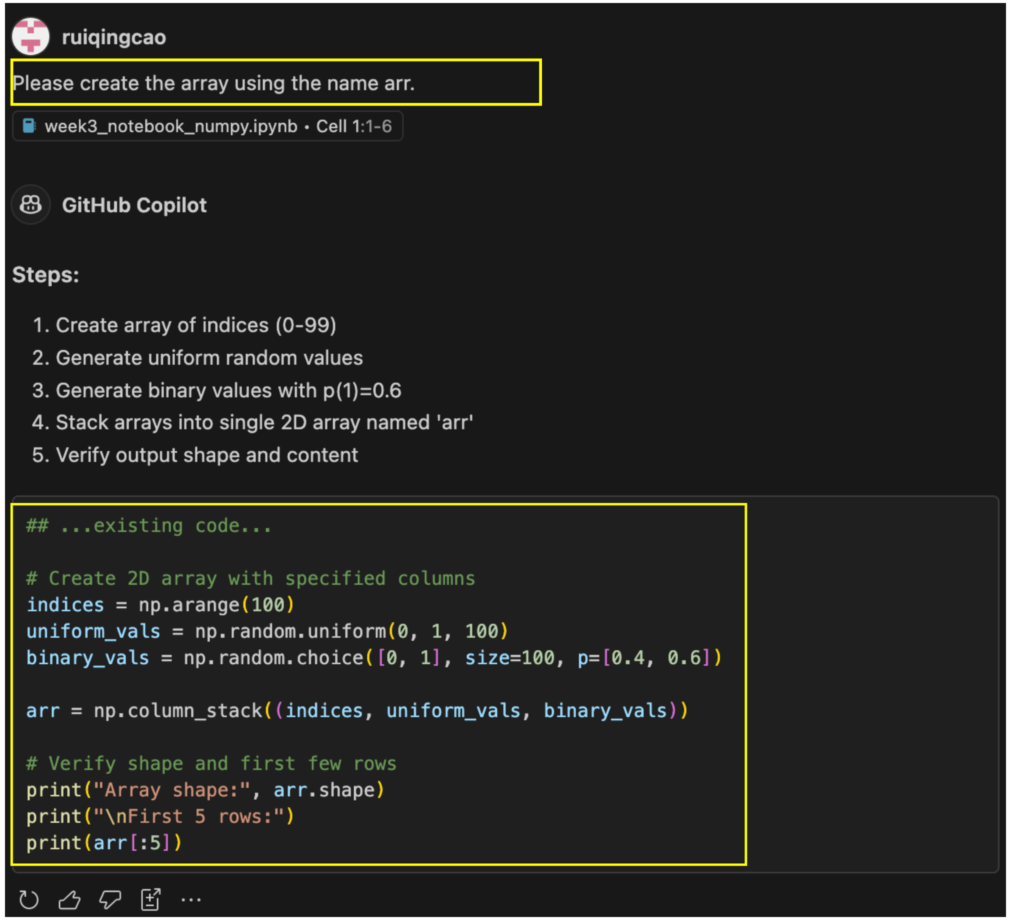
To run:

1. Open terminal (Cmd + `)
2. Run: `python numpy_array.py`

Using Generate AI to Generate Code

Ask for code adjustments

- “Please create the array using the name arr.”
- Generates Python code again to **meet the new requirement**



ruiqingcao

Please create the array using the name arr.

week3_notebook_numpy.ipynb • Cell 1:1-6

GitHub Copilot

Steps:

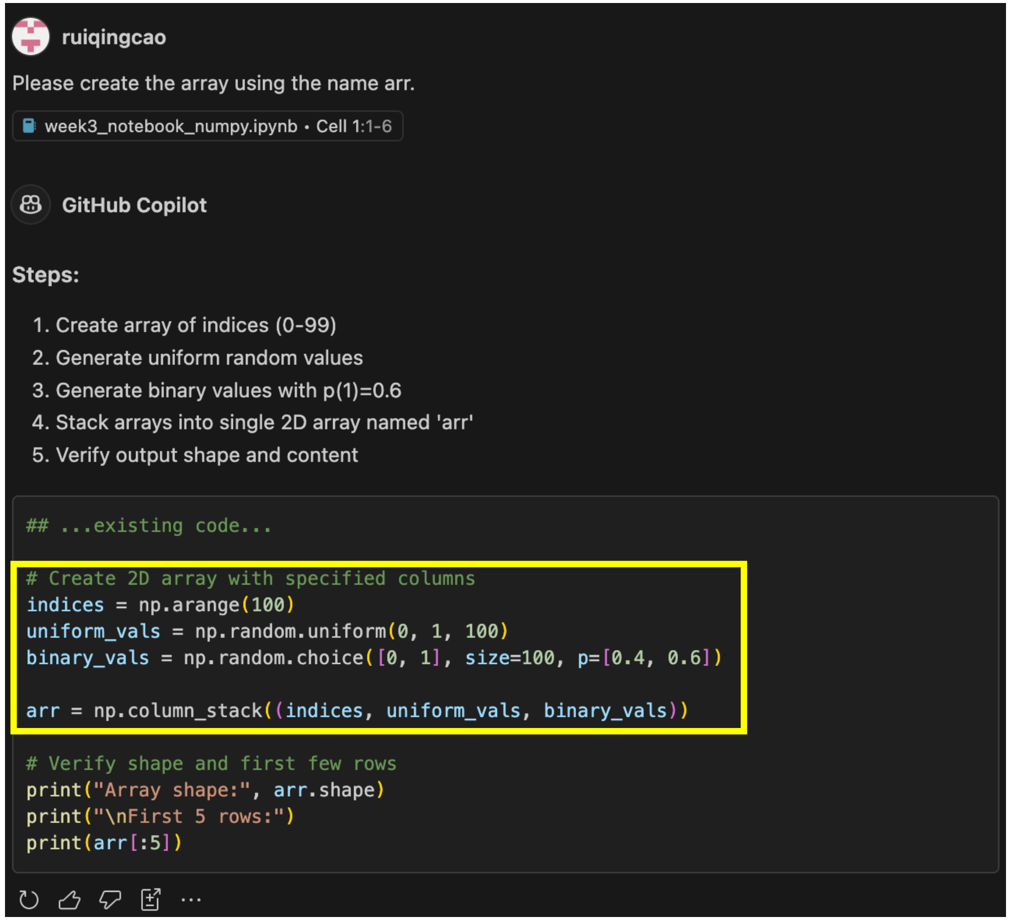
1. Create array of indices (0-99)
2. Generate uniform random values
3. Generate binary values with p(1)=0.6
4. Stack arrays into single 2D array named 'arr'
5. Verify output shape and content

```
## ...existing code...

# Create 2D array with specified columns
indices = np.arange(100)
uniform_vals = np.random.uniform(0, 1, 100)
binary_vals = np.random.choice([0, 1], size=100, p=[0.4, 0.6])

arr = np.column_stack((indices, uniform_vals, binary_vals))

# Verify shape and first few rows
print("Array shape:", arr.shape)
print("\nFirst 5 rows:")
print(arr[:5])
```

Using Generate AI to Generate Code

Select generated code partially

- Sometimes you only need **part** (not all) of the generated code
- Understand the meaning of the code, and **select** the part of the code that does what you need
- For example, no need to print the first 5 rows of the array



ruiqingcao

Please create the array using the name arr.

GitHub Copilot

Steps:

1. Create array of indices (0-99)
2. Generate uniform random values
3. Generate binary values with p(1)=0.6
4. Stack arrays into single 2D array named 'arr'
5. Verify output shape and content

```
## ...existing code...

# Create 2D array with specified columns
indices = np.arange(100)
uniform_vals = np.random.uniform(0, 1, 100)
binary_vals = np.random.choice([0, 1], size=100, p=[0.4, 0.6])

arr = np.column_stack((indices, uniform_vals, binary_vals))

# Verify shape and first few rows
print("Array shape:", arr.shape)
print("\nFirst 5 rows:")
print(arr[:5])
```

Indexing a Numpy Array

- Recall that `list` is a Python built-in type: a NumPy array is very similar to a `list`, but has more capabilities and computes faster

- Recall that indexes **start at 0**: this applies to *all array-like data objects*, including NumPy arrays

➢Index of a 1D array a: `a[n]` selects the element located at n (in a)

➢Index of a 2D array b: `b[m][n]` or `b[m,n]` selects the element located at row m and column n (in b)

Slicing a NumPy Array

General syntax: `array[start:end:step]` (similar to `list`)

Slice the 1D array a:

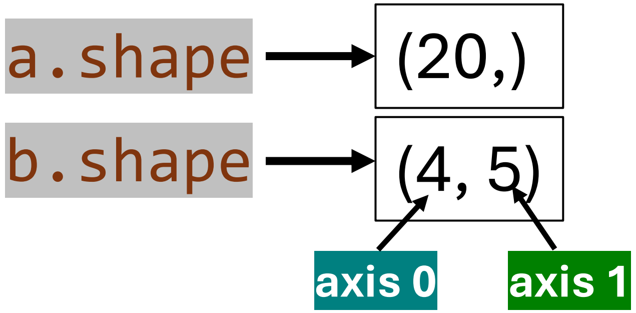
- Select elements from *index n to m*: `a[n:m+1]`
- Select elements from *index n to the end*: `a[n:]`
- Select the *first n elements* (from `0` to `n-1`): `a[:n]`

Slice the 2D array b:

- Select *row m* as a 1D array: `b[m,:]`
- Select *column n* as a 1D array: `b[:,n]`

Size of a NumPy Array

- The `ndarray.shape` attribute returns the dimension of an array as a tuple. Suppose a is a 1D array of length 20, and b is a 2D array with 4 rows and 5 columns



`a.shape` → (20,)

`b.shape` → (4, 5)

- Alternatively, you can use the built-in `len()` function to return the number of elements along the first dimension (axis 0) of an object

`len(a)` → 20

`len(b)` → 4

Sorting a Numpy Array

The `np.sort()` function can be used to sort arrays

- Sort a <u>1D array</u> a:

`np.sort(a)`

- Sort a <u>2D array</u> b on its values in <u>column j</u> (note: `np.lexsort()` takes a **<u>list</u>** of columns to sort on from least to most important)

`np.lexsort([b[:,j]])`

Note: `np.sort()` always sorts numbers in ascending order; To reverse the order, simply apply slicing after sorting: e.g., `np.sorted(a)[::-1]`

Unique Values & Frequencies Counts

- A very common way to explore a particular variable in a data set is to look at its range (i.e., unique values that it can possibly take)

```
np.unique(a)
```

...returns the unique values in the 1D array **a**

- Relatedly, we may also want to know the frequency of each value

```
np.unique(a, return_counts=True)
```

...returns unique values and their frequency counts in the 1D array **a**

Summary Statistics of Data in 2D Array

A 2D NumPy array `b` stores tabular data, with rows as observations and columns as variables. To compute summary statistics for each variable, aggregate the array along rows (axis=0) using aggregation functions like mean, standard deviation, median, max, or min

Statistic	Aggregation Function	Robust to Missing Values
Mean	np.mean(b,axis=0)	
StDev	np.std(b,axis=0)	
Median	np.median(b,axis=0)	
Maximum	np.max(b,axis=0)	
Minimum	np.min(b,axis=0)	
Sum	np.sum(b,axis=0)	

Problem: returns `np.nan` if there is even one missing value in a variable

Summary Statistics of Data in 2D Array

A 2D NumPy array `b` stores tabular data, with rows as observations and columns as variables. To compute summary statistics for each variable, aggregate the array along rows (axis=0) using aggregation functions like mean, standard deviation, median, max, or min

Statistic	Aggregation Function	Robust to Missing Values
Mean	np.mean(b,axis=0)	np.nanmean(b,axis=0)
StDev	np.std(b,axis=0)	np.nanstd(b,axis=0)
Median	np.median(b,axis=0)	np.nanmedian(b,axis=0)
Maximum	np.max(b,axis=0)	np.nanmax(b,axis=0)
Minimum	np.min(b,axis=0)	np.nanmin(b,axis=0)
Sum	np.sum(b,axis=0)	np.nansum(b,axis=0)

Problem: returns `np.nan` if there is even one missing value in a variable

Plot Frequency Histogram for a Variable

- Import the visualization library first

```
from matplotlib import pyplot as plt
```

- The underlying NumPy function `np.histogram()` generates frequency histograms, but for plotting, it does not need to be directly called. Instead

```
plt.hist(array[:,j])
plt.show()
```

…plots the frequency histogram of the variable in `array`’s column `j`

Exercise: NumPy Array Basic Operations

1. Create *two* 2D NumPy arrays X1 and X2 *independently* using the previous exercise with 100 rows and 3 columns *(column 0: 0-99; column 1: Uniform[0,1], column 2: Bernoulli(0.6))*
2. Compute the difference between X1 and X2, square each element of the result, and store the new array in Y
   - ➢ *Hint: matrix operations work on NumPy arrays (though they don't work on lists)*
3. Sort rows of Y on the values of column 1 (*hint: use `np.lexsort()`*)
4. Compute the unique value counts for Y's column 2
5. Slice rows 50–99 of columns 1 and 2 from Y into a new array Y2
6. For each column of Y2, compute the mean, median, and standard deviation, and plot the frequency histograms