经济大数据与 Python 应用

Introduction to NumPy

陈洲
湘潭大学商学院
2025

Press Space for next page →

Contents

Introduction to NumPy

Why Numpy

Datasets: fundamentally arrays of numbers.

NumPy (short for *Numerical Python*)

- 🧑‍🎨 Efficient storage
- 👷‍♀️ manipulation of numerical arrays

Installation

- Go to http://www.numpy.org/ and follow the installation instructions found there.

```
$ conda install numpy
```

- Import NumPy and double-check the version: Demo

In terminal

Start jupyter notebook

```
jupyter lab
```

In notebook

```
import numpy
numpy.__version__
```

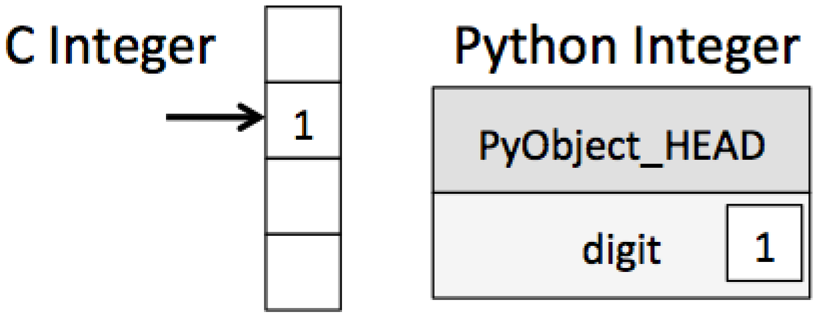
By convention, import NumPy using `np` as an alias:

```
import numpy as np
```

Understanding Data Types in Python

A Python Integer Is More Than Just an Integer

- Overhead involved in storing an integer in Python
- Analogy: 📦 Mail package



`PyObject_HEAD` is the part of the structure

A Python List Is More Than Just a List

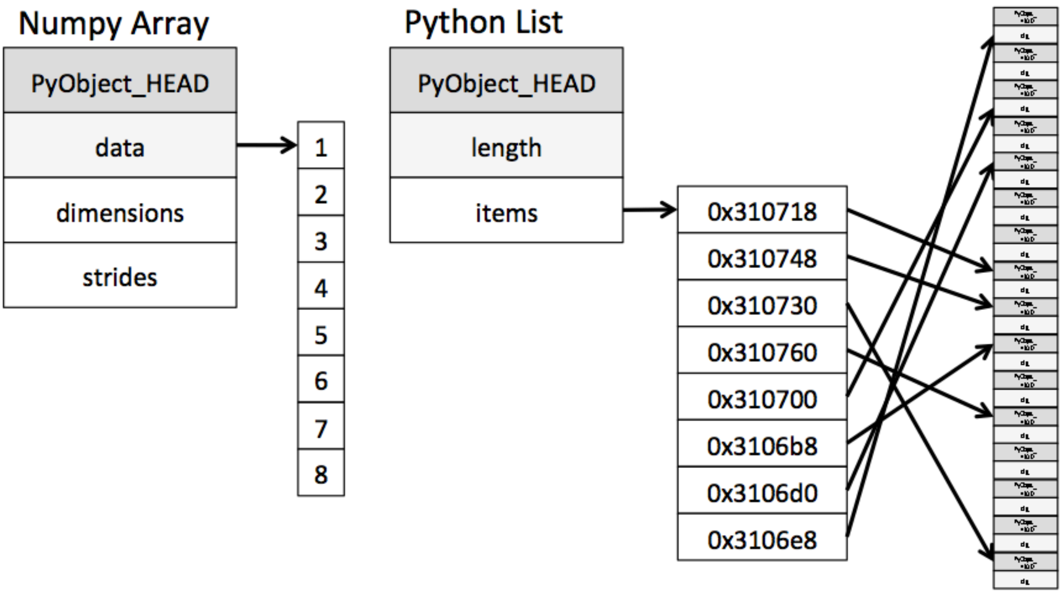
- Holds many Python objects.

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
```

This flexibility comes at a $ cost:

- Each item in the list must **contain its own type, reference count, and other information**.
- In the special case that all variables are of the same type
  - much of this information is **redundant**

The difference between a dynamic-type list and a fixed-type (NumPy-style) array



Fixed-type NumPy-style arrays are much more ✨ **efficient for storing and manipulating data**.

Creating Arrays from Python Lists

We'll start with the standard NumPy import, under the alias `np`:

Demo

```
import numpy as np
```

Use `np.array` to create arrays from Python lists:

```
# Integer array
np.array([1, 4, 2, 5, 3])
```

NumPy arrays can only contain data of the ❗ **same type**.

- If the types do not match, NumPy will **upcast** them according to its type promotion rules
- Integers are upcast to floating point:

```
np.array([3.14, 4, 2, 3])
```

Explicitly set the data type of the resulting array

- Use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype=np.float32)
```

NumPy arrays can be ✨ **multidimensional**

Initializing a multidimensional array using a list of lists:

```
# Nested lists result in multidimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

For larger arrays, it is more efficient

- To create arrays from scratch using routines built into NumPy.

Here are several examples:

```
# Create a length-10 integer array filled with 0s
np.zeros(10, dtype=int)
```

```
# Create a 3×5 floating-point array filled with 1s
np.ones((3, 5), dtype=float)
```

```
# Create a 3×5 array filled with 3.14
np.full((3, 5), 3.14)
```

✨ Number Series

```
# Create an array filled with a linear sequence
# starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range function)
np.arange(0, 20, 2)
```

```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

✨ Random Array

```
# Create a 3×3 array of uniformly distributed
# pseudorandom values between 0 and 1
np.random.random((3, 3))
```

```
# Create a 3×3 array of normally distributed pseudorandom
# values with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

```
# Create a 3×3 array of pseudorandom integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

✨ Special Matrix

```
# Create a 3×3 identity matrix
np.eye(3)
```

```
# Create an uninitialized array of three integers; the values will be
# whatever happens to already exist at that memory location
np.empty(3)
```

The Basics of NumPy Arrays

Data **manipulation** in Python -> NumPy array ✨ manipulation:

- Pandas Part 3 are built around the NumPy array

NumPy array manipulation

- subarrays
- split
- reshape
- join the arrays.

NumPy Array Attributes

NumPy's **random number generator**

- *seed* with a set value in order
- ensure that the **same random arrays** are generated each time this code is run:

```
import numpy as np
rng = np.random.default_rng(seed=1701)  # seed for reproducibility
```

Define random arrays of

- One, two, and three dimensions.

```
# Return random integers from low (inclusive, 0) to high (exclusive)
x1 = rng.integers(10, size=6)  # one-dimensional array
x2 = rng.integers(10, size=(3, 4))  # two-dimensional array
x3 = rng.integers(10, size=(3, 4, 5))  # three-dimensional array
```

💪 Recall

How to generate random integers between 0 and 10 using `np.random`.

Each array has attributes

- `ndim` (the number of dimensions)
- `shape` (the size of each dimension)
- `size` (the total size of the array)
- `dtype` (the type of each element):

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:   ", x3.dtype)
```

Array Indexing: Accessing Single Elements

The i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets

```
x1
x1[0]
x1[4]
```

To index from the end of the array, you can use negative indices:

```
x1[-1]
x1[-2]
```

Multidimensional array

- using a comma-separated `(row, column)` tuple:

```
x2
x2[0,0]
x2[2,0]
x2[2,-1]
```

Values can also be ✨ **modified** using any of the preceding index notation:

```
x2[0, 0] = 12
x2
```

💪 Task

Generate a matrix m of integers (two-dimensional array) with shape $(5, 5)$

- for element $m_{ij} \in m$ at (i, j), $m_{ij} = i * j$

```
m = np.zeros((5,5))
for i in range(5):
  for j in range(5):
    m[i,j] = i * j
m
```

Caution: NumPy arrays have a fixed type.

- insert a floating-point value into an integer array
- the value will be 👎 *silently truncated*

Array Slicing: Accessing Subarrays

Access subarrays with the *slice* notation

- the colon (`:`) character.

```
x[start:stop:step]
```

Default values `start=0` , `stop=<size of dimension>` , `step=1` .

```
# Question
a = np.array([1,2,3])
a[0:2]
```

One-Dimensional Subarrays

```
x1
x1[:3]  # first three elements
x1[3:]  # elements after index 3
x1[1:4]  # middle subarray
x1[::2]  # every second element
x1[1::2]  # every second element, starting at index 1
```

❓ Question: negative step

```
x1[::-1]
x1[4::-2]
```

Multidimensional Subarrays

```
x2
x2[:2, :3]  # first two rows & three columns
x2[:3, ::2]  # three rows, every second column
x2[::-1, ::-1]  # all rows & columns, reversed
```

Accessing array rows and columns

Accessing single rows or columns of an array.

- Single colon (`:`):

```
x2[:, 0]  # first column of x2
x2[0, :]  # first row of x2
```

The 👉 **empty slice** can be omitted for a more compact syntax:

```
x2[0]  # equivalent to x2[0, :]
```

Subarrays as ❎ **No-Copy Views**

NumPy array slices are returned as *views*

- rather than *copies* of the array data.

```
print(x2)
```

Let’s extract a 2×2 subarray from this:

```
x2_sub = x2[:2, :2]
print(x2_sub)
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
x2_sub[0, 0] = 99
print(x2_sub)
print(x2)
```

Creating Copies of Arrays

Explicitly copy the data within an array or a subarray.

- `copy` method:

```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

If we now modify this subarray, the original array is not touched:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
print(x2)
```

💪 Task

Find all even numbers under 100 as a NumPy array.

```
numbers = np.arange(100)
numbers[::2]
```

Reshaping of Arrays

Put the numbers 1 through 9 in a 3×3 grid:

```
grid = np.arange(1, 10).reshape(3, 3)
print(grid)
```

- `reshape` method will return a **no-copy view** of the initial array.

✨ One-dimensional array to row or column matrix

```
x = np.array([1, 2, 3])
x.reshape((1, 3))  # row vector via reshape
x.reshape((3, 1))  # column vector via reshape
```

Shorthand for this is to use `np.newaxis` in the slicing syntax:

```
x[np.newaxis, :]  # row vector via newaxis
x[:, np.newaxis]  # column vector via newaxis
```

Array Concatenation and Splitting

- Combine multiple arrays into one
- Split a single array into multiple arrays.

Concatenation of Arrays

- `np.concatenate`
- `np.vstack`
- and `np.hstack`

`np.concatenate` takes a tuple or list of arrays as its first argument, as you can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

You can also concatenate more than two arrays at once:

```
z = np.array([99, 99, 99])
print(np.concatenate([x, y, z]))
```

And it can be used for two-dimensional arrays:

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

```
# concatenate along the first axis
np.concatenate([grid, grid])
```

```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

Clearer:

- `np.vstack` (vertical stack)
- `np.hstack` (horizontal stack) functions:

```
# vertically stack the arrays
np.vstack([x, grid])
```

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Splitting of Arrays

- `np.split`
- `np.hsplit`
- `np.vsplit`

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

The related functions `np.hsplit` and `np.vsplit` are similar:

```
grid = np.arange(16).reshape((4, 4))
grid
```

```python
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```python
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

💪 Task

```
import numpy as np

# Monthly inflation rates for 4 years (48 months)
inflation_data = np.array([
    2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2,
    2.0, 1.9, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3, 1.2, 1.1, 1.0, 0.9,
    0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0, -0.1, -0.2, -0.3,
    -0.4, -0.5, -0.6, -0.7, -0.8, -0.9, -1.0, -1.1, -1.2, -1.3, -1.4, -1.5
])
```

- Extract data for the third year (25 to 36)

```
third_year_inflation_data = inflation_data[24:36]
```

- Split data to years (12, 24, 36)

```
np.split(inflation_data, [12, 24, 36])
```

Computation on NumPy Arrays: Universal Functions

NumPy arrays can be very fast, or it can be very slow.

- making it fast is to use vectorized operations
- implemented through NumPy's *universal functions* (ufuncs)

The Slowness of Loops

Situations where many small operations are being repeated

- looping over arrays to operate on each element.

For example,

- an array of values
- compute the reciprocal of each.

```
nums = np.array([1, 2, 3])
```

A straightforward approach might look like this:

```
import numpy as np
rng = np.random.default_rng(seed=1701)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output

values = rng.integers(1, 10, size=5)
compute_reciprocals(values)
```

Benchmark this with IPython's `%timeit` magic

```
big_array = rng.integers(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

Introducing Ufuncs

NumPy provides a *vectorized* operation.

Compare the results of the following two operations:

```
print(compute_reciprocals(values))
print(1.0 / values)
```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```
%timeit (1.0 / big_array)
```

Operate between two arrays:

```
np.arange(5) / np.arange(1, 6)
```

Any time you see such a *loop* in a NumPy script

- 👉 **replaced with a vectorized expression**

Exploring NumPy’s Ufuncs

Array Arithmetic

```
x = np.arange(4)
print("x      =", x)
print("x + 5  =", x + 5)
print("x - 5  =", x - 5)
print("x * 2  =", x * 2)
print("x / 2  =", x / 2)
print("x // 2 =", x // 2)  # floor division
```

- A `**` operator for exponentiation,
- A `%` operator for modulus:

```
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
```

The standard order of operations is respected:

```
-(0.5*x + 1) ** 2
```

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Absolute Value

Just as NumPy understands Python's built-in arithmetic operators,

- it also understands Python's built-in absolute value function:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

The corresponding NumPy ufunc is `np.absolute`, which is also available under the alias `np.abs`:

```
np.absolute(x)
np.abs(x)
```

This ufunc can also handle complex data, in which case it returns the magnitude:

```
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
```

Trigonometric Functions

Start by defining an array of angles:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

Inverse trigonometric functions are also available:

```
x = [-1, 0, 1]
print("x         = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
```

Exponents and Logarithms

Other common operations available in NumPy ufuncs are the exponentials:

```
x = [1, 2, 3]
print("x   =", x)
print("e^x =", np.exp(x))
print("2^x =", np.exp2(x))
print("3^x =", np.power(3., x))
```

The basic `np.log` gives the natural logarithm

- base-2 logarithm or the base-10 logarithm

```
x = [1, 2, 4, 10]
print("x        =", x)
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

Some specialized versions that are useful for maintaining precision with very small input:

```
x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))
```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were to be used.

Aggregations: min, max, and Everything in Between

A first step in exploring any dataset

- Summary statistics
- Mean and standard deviation
- Median, minimum and maximum, quantiles, etc.

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and try out some of them here.

Summing the Values in an Array

Your 🍀 lucky numbers

Or random numbers

```python
import numpy as np
rng = np.random.default_rng()
```

```python
L = rng.random(100)
sum(L)
```

NumPy’s `sum` function, and the result is the same in the simplest case:

```
np.sum(L)
```

NumPy’s version of the operation is computed much more quickly:

```
big_array = rng.random(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

Minimum and Maximum

Python has built-in `min` and `max` functions

```
min(big_array), max(big_array)
```

NumPy’s corresponding functions have similar syntax

- again operate much more quickly:

```
%timeit min(big_array)
%timeit np.min(big_array)
```

A shorter syntax is to use methods of the array object itself:

```
print(big_array.min(), big_array.max(), big_array.sum())
```

Whenever possible, make sure

- using the NumPy version of these aggregates when operating on NumPy arrays!

Multidimensional Aggregates

Data stored in a two-dimensional array:

```
M = rng.integers(0, 10, (3, 4))
print(M)
```

NumPy aggregations will apply across all elements of a multidimensional array:

```
M.sum()
```

Find the minimum value within each column by specifying `axis=0`:

```
M.min(axis=0)
```

Find the maximum value within each row:

```
M.max(axis=1)
```

The `axis` keyword specifies the dimension of the array that will be *collapsed*, rather than the dimension that will be returned.

- `axis=0` means that axis 0 will be collapsed: for two-dimensional arrays, values within each column will be aggregated.

💪 Task: OLS

For a simple linear regression model

$$y = \beta_0 + \beta_1 x + u$$

- `x = np.random.randint(0, 100, size=(100,))`
- `y = np.random.randint(0, 100, size=(100,))`
- Get OLS estimator $\hat{\beta}_1 = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$

Task: Discrete choice (optional)

The optimal daily studying hours h^* for student type c is given by

$$h^*(c) = \underset{h \in \{0,1,...,8\}}{\arg\max}\ u(c,h), c \in \{3,5,7\}$$

where $u(c,h) = \log(h) - 0.2(h-c)^2$.

Find the optimal daily studying hours `h_star(c)` for students of type c.

```python
def u(c, h):
    return np.log(h+1) - 0.2*(h - c)**2
```

```python
def h_star(c):
    h_array = np.arange(0, 9)
    max_h = None
    max_utility = -np.inf
    for h in h_array:
        utility = u(c, h)
        if utility > max_utility:
            max_h = h
            max_utility = utility
    return max_h
```

```python
def h_star2(c):
    h_array = np.arange(0, 9)
    index = np.argmax(u(c, h_array))
    return h_array[index]
```

Computation on Arrays: Broadcasting

Introducing Broadcasting

Binary operations are performed on an **element-by-element** basis:

```python
import numpy as np
```

```python
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

Arrays of different sizes — add a scalar (think of it as a zero-dimensional array) to an array:

```
a + 5
```

Higher dimension.

- Observe the result when we add a one-dimensional array to a two-dimensional array:

```
M = np.ones((3, 3))
M
```

```
M + a
```

`a` is stretched, or broadcasted, across the second dimension in order to match the shape of `M`

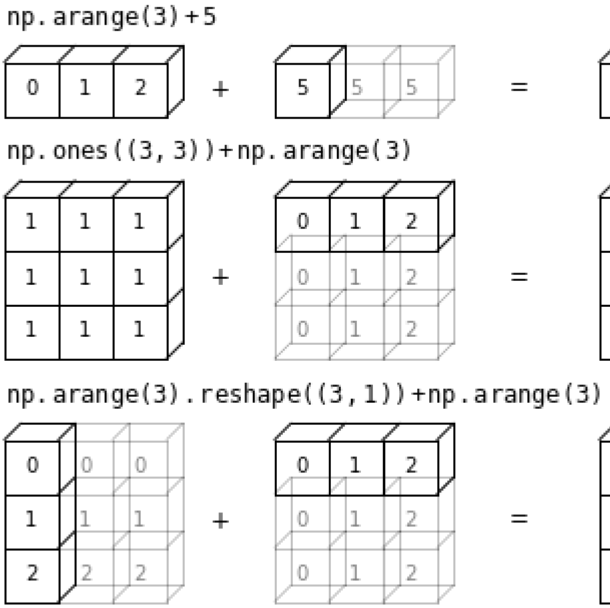
Broadcasting of both arrays. Consider the following example:

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)
```

```
a + b
```

Stretched or broadcasted one value to match the shape of the other



Broadcasting Example 1

Suppose we want to add a two-dimensional array to a one-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)
```

- `M.shape` is `(2, 3)`
- `a.shape` is `(3,)`

The array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape` remains `(2, 3)`
- `a.shape` becomes `(1, 3)`

The first dimension disagrees, so we stretch this dimension to match:

- `M.shape` remains `(2, 3)`
- `a.shape` becomes `(2, 3)`

The shapes now match, and we see that the final shape will be `(2, 3)`:\

```
M + a
```

Because the results match, these shapes are compatible. We can see this here:

```
a + b
```

Broadcasting Example

Next, let's take a look at an example in which the two arrays are 🙄 not compatible:

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape` is `(3, 2)`
- `a.shape` is `(3,)`

We must pad the shape of `a` with ones:

- `M.shape` remains `(3, 2)`
- `a.shape` becomes `(1, 3)`

The first dimension of `a` is then stretched to match that of `M`:

- `M.shape` remains `(3, 2)`
- `a.shape` becomes `(3, 3)`

Now we hit rule 3—the final shapes do not match, so these two arrays are **incompatible**, as we can observe by attempting this operation:

```
M + a
```

Centering an Array

```
rng = np.random.default_rng(seed=1701)
X = rng.random((10, 3))
```

Compute the mean of each column using the `mean` aggregate across the first dimension:

```
Xmean = X.mean(0)
Xmean
```

Center the `X` array by subtracting the mean (this is a broadcasting operation):

```
X_centered = X - Xmean
```

Check that the centered array has a mean near zero:

```
X_centered.mean(0)
```

To within machine precision, the mean is now zero.

Task

In a virtual world, the probability that a graduate in economics finds a job with a salary of x is respectively

$$
\begin{aligned}
\Pr(x = 3000) &= 0.1 \\
\Pr(x = 5000) &= 0.3 \\
\Pr(x = 7000) &= 0.3 \\
\Pr(x = 9000) &= 0.2 \\
\Pr(x = 20000) &= 0.1
\end{aligned}
$$

Given

```
x = np.array([3000, 5000, 7000, 9000, 20000])
pr = np.array([0.1, 0.3, 0.3, 0.2, 0.1])
```

Find the expected wage.

Task

Multiplication table

```
1×1=1
2×1=2 2×2=4
3×1=3 3×2=6 3×3=9
4×1=4 4×2=8 4×3=12 4×4=16
5×1=5 5×2=10 5×3=15 5×4=20 5×5=25
6×1=6 6×2=12 6×3=18 6×4=24 6×5=30 6×6=36
7×1=7 7×2=14 7×3=21 7×4=28 7×5=35 7×6=42 7×7=49
8×1=8 8×2=16 8×3=24 8×4=32 8×5=40 8×6=48 8×7=56 8×8=64
9×1=9 9×2=18 9×3=27 9×4=36 9×5=45 9×6=54 9×7=63 9×8=72 9×9=81
```

Comparisons, Masks, and Boolean Logic

Comparison Operators as Ufuncs

Computation on NumPy Arrays: Universal Functions introduced ufuncs

- `+`, `-`, `*`, `/`: element-wise operations.
- `<` and `>` as element-wise ufuncs.
- The result of these comparison: Boolean array.

```python
x = np.array([1, 2, 3, 4, 5])
x < 3  # less than
x > 3  # greater than
x <= 3  # less than or equal
x >= 3  # greater than or equal
x != 3  # not equal
x == 3  # equal
```

It is also possible to do an element-wise comparison of two arrays, and to include compound expressions:

```
(2 * x) == (x ** 2)
```

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
rng = np.random.default_rng(seed=1701)
x = rng.integers(10, size=(3, 4))
x
```

```
x < 6
```

Working with Boolean Arrays

The two-dimensional array we created earlier:

```
print(x)
```

Counting Entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

```
# how many values less than 6?
np.count_nonzero(x < 6)
```

`False` is interpreted as `0`, and `True` is interpreted as `1`:

```
np.sum(x < 6)
```

The benefit of `np.sum` is that this summation can be done along rows or columns as well:

```
# how many values less than 6 in each row?
np.sum(x < 6, axis=1)
```

Checking whether any or all the values are `True`, we can use (you guessed it) `np.any` or `np.all`:

```
# are there any values greater than 8?
np.any(x > 8)
```

```
# are there any values less than zero?
np.any(x < 0)
```

```
# are all values less than 10?
np.all(x < 10)
```

```
# are all values equal to 6?
np.all(x == 6)
```

`np.all` and `np.any` can be used along particular axes as well.

```
# are all values in each row less than 8?
np.all(x < 8, axis=1)
```

Boolean Arrays as Masks

- To select particular subsets of the data themselves.

Let's return to our `x` array from before:

```
x
```

Suppose we want an array of all values in the array that are less than 5.

- We can obtain a Boolean array for this condition

```
x < 5
```

To *select* these values from the array

- a *masking* operation:

```
x[x < 5]
```

Al the values in positions at which the mask array is `True` s.

Using the Keywords and/or Versus the Operators `&` and `|`

- `and` and `or` operate on the object as a whole
- `&` and `|` operate on the elements within the object.

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
```

But if you use `or` on these arrays it will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
# raise ValueError
A or B
```

Similarly, when evaluating a Boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
x = np.arange(10)
(x > 4) & (x < 8)
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw previously:

```
# raise ValueError
(x > 4) and (x < 8)
```

Remember this:

- `and` and `or` perform a single Boolean evaluation on an entire object
- while `&` and `|` perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object.

For Boolean NumPy arrays, the latter is nearly always the desired operation

Fancy Indexing

Before: to access and modify portions of arrays

- using simple indices (e.g., `arr[0]`),
- slices (e.g., `arr[:5]`)
- and Boolean masks (e.g., `arr[arr > 0]`).

In this chapter, another style of array indexing

Exploring Fancy Indexing

Fancy indexing

- passing an array of indices to access multiple array elements at once

For example, consider the following array:

```
import numpy as np
rng = np.random.default_rng(seed=1701)

x = rng.integers(100, size=10)
print(x)
```

Suppose we want to access three different elements

```
[x[3], x[7], x[2]]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
ind = [3, 7, 4]
x[ind]
```

When using arrays of indices, the shape of the result reflects the shape of the *index arrays*

```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
X = np.arange(12).reshape((3, 4))
X
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

Combined Indexing

Fancy indexing can be combined with the other indexing schemes

For example, given the array `X`:

```
print(X)
```

We can combine fancy and simple indices:

```
X[2, [2, 0, 1]]
```

To slicing several rows:

- We can also combine fancy indexing with slicing:

```
X[1:, [2, 0, 1]]
```

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
```

Task

Simulate a Gacha game with four different rarities: 999, 99, and 9, each with the following probabilities:

$$\begin{gathered}\Pr(999) = 0.01\\ \Pr(99) = 0.10\\ \Pr(9) = 0.89\end{gathered}$$

Perform a simulation of 100 pulls

- keeping track of the count for each rarity

Sorting Arrays

Fast Sorting in NumPy: np.sort and np.argsort

The `np.sort` function is analogous to Python's built-in `sorted` function

- efficiently return a sorted copy of an array:

```
import numpy as np

x = np.array([2, 1, 4, 3, 5])
np.sort(x)
```

Similarly to the `sort` method of Python lists, you can also sort an array in-place using the array `sort` method:

```
x.sort()
print(x)
```

A related function is `argsort`

- returns the *indices* of the sorted elements:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

The first element of this result gives the index of the smallest element

- the second value gives the index of the second smallest, and so on.
- These indices can then be used (via fancy indexing) to construct the sorted array if desired:

```
x[i]
```

Sorting Along Rows or Columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns

- using the `axis` argument. For example:

```
rng = np.random.default_rng(seed=42)
X = rng.integers(0, 10, (4, 6))
print(X)
```

```
# sort each column of X
np.sort(X, axis=0)
```

```
# sort each row of X
np.sort(X, axis=1)
```

Partial Sorts: Partitioning

To find the *k* smallest values in the array

- the `np.partition` function.
- `np.partition` takes an array and a number *K*
- the result is a new array with the smallest *K* values to the left of the partition and the remaining values to the right:

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
```

Notice that the first three values in the resulting array are the three smallest in the array

- the remaining array positions contain the remaining values.
- Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

```
np.partition(X, 2, axis=1)
```

Task

GDP data for 5 countries (in billions USD)

```
gdp_data = np.array([2500, 3000, 1500, 4000, 3500])
```

Asian countries indicator

```
asian = np.array([1, 0, 0, 1 0)]
```

- Top gdp among asian countries.
- Top gdp among non-asian countries.

END