DATA SCIENCE

Introduction to NumPy

Arrays, numerical operations, and a foundation for data science tools.

UNC
GREENSBORO

Why NumPy?

Suppose you need to multiply one million numbers by 2.

PYTHON LIST

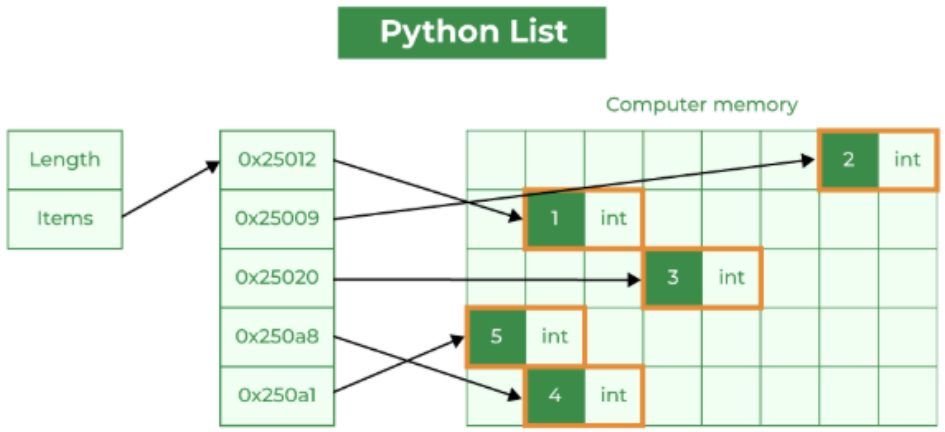
```
numbers = [1, 2, 3, 4, 5]
result = []
for x in numbers:
    result.append(x * 2)
```

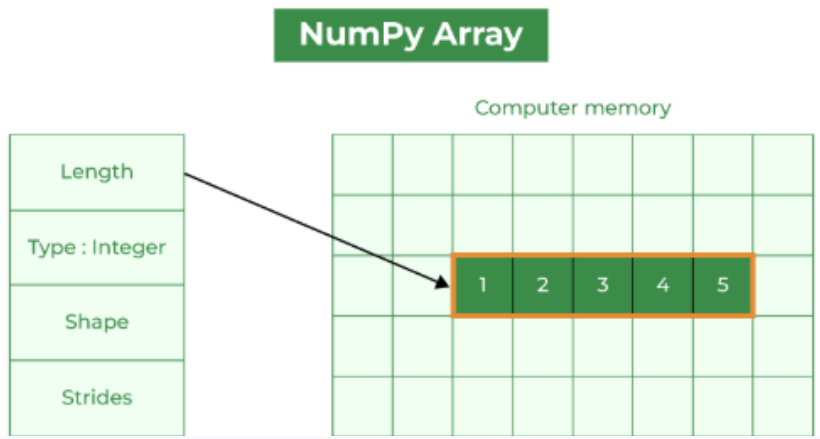
NUMPY

```
import numpy as np
numbers = np.array([1, 2, 3, 4, 5])
result = numbers * 2
```

Python list vs NumPy array

Python list	NumPy array
General-purpose container	Numerical data structure
Can mix data types	Usually uses one data type
Flexible structure	Structured array
Loops often handle each element	Vectorized operations work across elements
Nested lists for multiple dimensions	Native multidimensional arrays





What happens when we add 10?

Python list

A list does not define addition with a single number.

NumPy array

NumPy applies the operation to each element.

PYTHON

```
numbers = [1, 2, 3, 4]
numbers + 10
```

NUMPY

```
numbers = np.array([1, 2, 3, 4])
numbers + 10
```

RESULT

TypeError

RESULT

array([11, 12, 13, 14])

The + operator depends on the data structure

PYTHON LIST

```
a = [1, 2, 3]
b = [4, 5, 6]
a + b
```

CONCATENATES

[1, 2, 3, 4, 5, 6]

NUMPY ARRAY

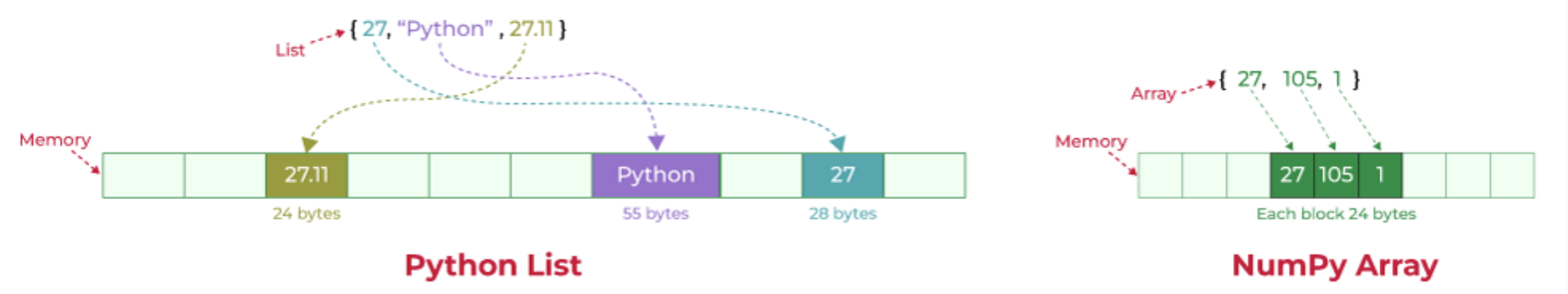
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
a + b
```

ADDS ELEMENT BY ELEMENT

[5 7 9]

Why NumPy is faster

NumPy arrays are optimized for numerical computations, with efficient element-wise operations and mathematical functions. **These operations are implemented in C**, resulting in faster performance than equivalent operations on lists.



Importing NumPy

STANDARD IMPORT

```
import numpy as np
```

A standard alias used throughout data science code.

The alias keeps common function calls easy to read.

Creating a 1D, 2D, 3D array

PYTHON

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

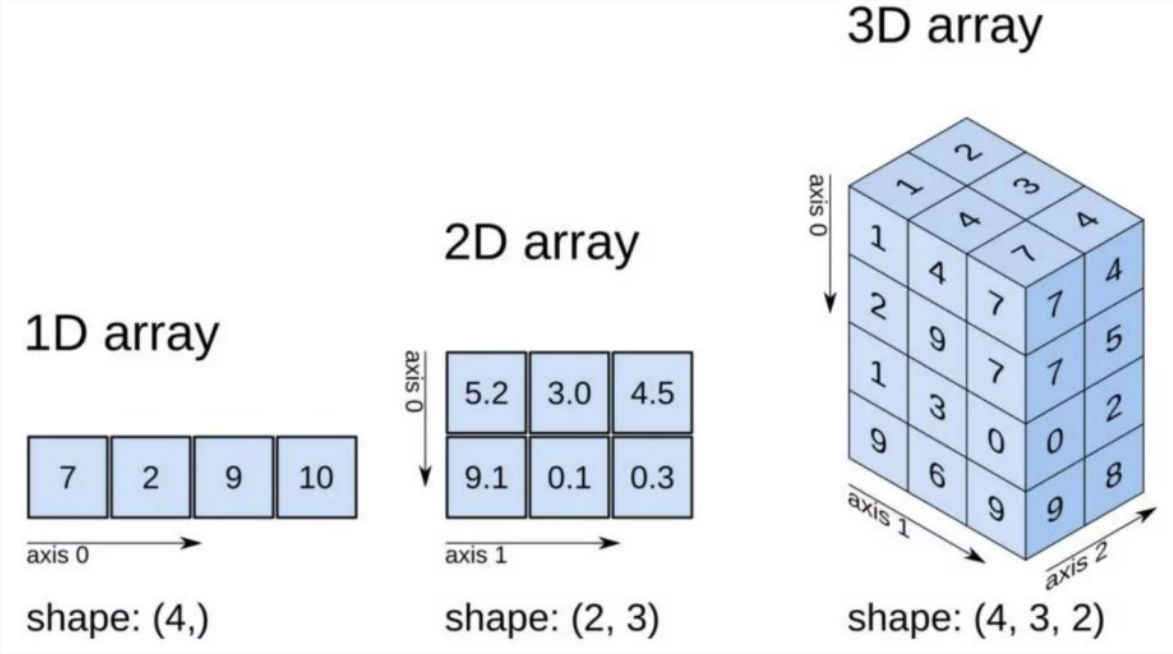
PYTHON

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

PYTHON

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

2 layers × 2 rows × 3 columns



ARRAY CREATION

Understanding array dimensions

EXAMPLE

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

a.ndim = 2

Number of dimensions

a.shape = (2, 3)

Size along each dimension

a.size = 6

Total number of elements

Read these three attributes before you manipulate an unfamiliar array.

Creating an empty array

Zero elements

np.array([]) creates an array with no elements.

PYTHON

```
np.array([])
```

Uninitialized values

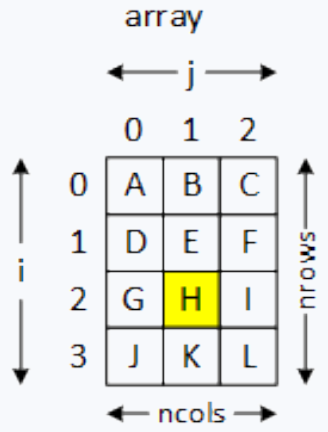
np.empty((2, 3)) allocates space without filling it with a known value.

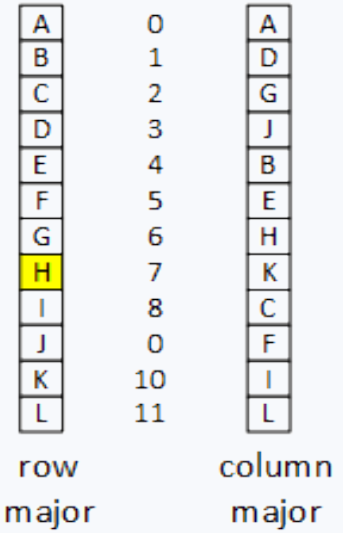
PYTHON

```
np.empty((2, 3))
```

```
numpy.empty(shape, dtype=float, order='C')
```

Parameter	Description
shape	Shape (dimensions) of the array
dtype	Data type of the array (optional)
order	Memory layout of the array: "C" = row-major order "F" = column-major order





Creating arrays of zeros

1D ARRAY

```
np.zeros(5)
```

2D ARRAY

```
np.zeros((2, 3))
```

Common uses

- Initialize numerical arrays
- Reserve placeholders for later results
- Set starting values for a calculation

```
numpy.zeros(shape, dtype=float, order='C')
```

Parameter	Description
shape	Shape (dimensions) of the array
dtype	Data type of the array (optional)
order	Memory layout of the array: "C" = row-major order (C-style) "F" = column-major order (Fortran-style)

Creating arrays of ones

1D ARRAY

```
np.ones(5)
```

2D ARRAY

```
np.ones((2, 3))
```

```
numpy.ones(shape, dtype=None, order='C')
```

Parameter	Description
shape	Shape (dimensions) of the array
dtype	Data type of the array (optional)
order	Memory layout of the array: "C" = row-major order (C-style) "F" = column-major order (Fortran-style)

Creating arrays with a range

EXAMPLE 1

```
np.arange(5)
```

OUTPUT

```
[0 1 2 3 4]
```

EXAMPLE 2

```
np.arange(2, 10)
```

OUTPUT

```
[2 3 4 5 6 7 8 9]
```

EXAMPLE 3

```
np.arange(0, 10, 2)
```

OUTPUT

```
[0 2 4 6 8]
```

np.arange(start, stop, step)

stop is not included

Evenly spaced values with np.linspace()

PYTHON

```
np.linspace(0, 1, 5)
```

0.00	0.25	0.50	0.75	1.00

np.arange()

You choose the step size.

np.linspace()

You choose the number of values between endpoints.

Changing shape with reshape()

START

a = np.arange(12)

12 ELEMENTS

[0 1 2 3 4 5 6 7 8 9 10 11]

RESHAPE

a.reshape(3, 4)

0	1	2	3
4	5	6	7
8	9	10	11

reshape() keeps the total number of elements the same.

Works

a.reshape(3, 4)
3 × 4 = 12

Does not work

a.reshape(5, 2)
5 × 2 = 10

Splitting arrays with np.split()

PYTHON

```
a = np.arange(12)
np.split(a, 3)
```

0	1	2	3

4	5	6	7

8	9	10	11

START

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

0	1	2	3
4	5	6	7
8	9	10	11

Split by rows

np.split(a, 3, axis=0)
axis=0 combines or splits within each column

Split by columns

np.split(a, 2, axis=1)
axis=1 combines or splits within each row

reshape() and resize()

reshape()

Changes the arrangement.
The total number of elements stays the same.

```
a = np.arange(6)
b = a.reshape(2, 3)
```

resize()

Changes the shape in place.
It can add or remove elements.

```
a = np.arange(6)
a.resize(2, 4)
```

If the new array is larger than the original array, the new array will contain repeated copies of the original elements.

Rounding with NumPy

PYTHON

```
a = np.array([1.234, 2.567, 3.891])
np.around(a, 2)
```

1.23	**2.57**	**3.89**

around() rounds each value to a chosen number of decimal places.

START

```
a = np.array([1.2, 1.8, 2.1, 2.9])
```

np.floor(a)

[1. 1. 2. 2.]
Moves toward negative infinity

np.ceil(a)

[2. 2. 3. 3.]
Moves toward positive infinity

Arithmetic with NumPy arrays

START

```
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])
```

Operation	Result
a + b	[11 22 33]
a − b	[9 18 27]
a * b	[10 40 90]
a / b	[10. 10. 10.]
a ** 2	[100 400 900]

NumPy arithmetic usually applies element by element.

Summarizing an array

DATA

```
a = np.array([10, 20, 30, 40, 50])
```

Function	Value	Meaning
np.sum(a)	150	Total
np.mean(a)	30	Average
np.min(a)	10	Smallest value
np.max(a)	50	Largest value
np.std(a)	14.14	Population standard deviation

From a NumPy array to a pandas Series/ DataFrame

PYTHON

```
import numpy as np
import pandas as pd

values = np.array([80, 90, 75, 88])
s = pd.Series(values)
```

SERIES

```
0    80
1    90
2    75
3    88
dtype: int64
```

NumPy array: values. pandas Series: values with an index.

PYTHON

```
data = np.array([[80, 90],
                 [75, 88],
                 [92, 85]])

df = pd.DataFrame(data,
                  columns=["Math", "Science"])
```

	Math	Science
0	80	90
1	75	88
2	92	85

Using NumPy functions with pandas

PANDAS METHOD

```
df["Math"].mean()
```

NUMPY FUNCTION

```
np.mean(df["Math"])
```

ANOTHER EXAMPLE

```
np.round(df["Math"], 1)
```

NumPy functions often work directly with pandas Series and DataFrames.