NumPy Fundamentals

1. What Is NumPy?

NumPy stands for **Numerical Python**.

It is a Python library designed for working efficiently with numerical data, especially large collections of values arranged in one or more dimensions.

Its central data structure is the **`ndarray`**, a homogeneous N-dimensional array. NumPy also provides functions for mathematical operations, statistics, array manipulation, linear algebra, random number generation, and much more.

Install it with:

```
pip install numpy
```

Import it using the conventional alias:

```
import numpy as np
```

The `np` convention is widely used throughout Python's scientific-computing ecosystem.

---

2. Why Do We Need NumPy?

Python lists are excellent general-purpose containers.

For example:

```
numbers = [10, 20, 30, 40]
```

But numerical applications often need to process:

```
thousands of measurements
millions of pixels
large datasets
mathematical matrices
scientific simulations
machine learning data
financial calculations
```

NumPy is designed for large quantities of **homogeneous numerical data** and provides efficient operations over entire arrays.

Consider:

```
numbers = [10, 20, 30]

result = [x * 2 for x in numbers]
```

With NumPy:

```
numbers = np.array([10, 20, 30])

result = numbers * 2
```

The second approach expresses the numerical operation directly:

```
array × number
     
every element × number
```

This style is called **vectorized array computation**.

---

3. NumPy's Core Mental Model

Think of NumPy as:

```
Python
  
NumPy
  
ndarray
  
Numerical operations
```

The most important object is:

```
np.ndarray
```

A NumPy array contains:

```
Data
+
Shape
+
Data type
+
Dimensions
```

For example:

```
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])
```

Visually:

```
    columns
    □ □ □

  10 20 30
  40 50 60
  □
 rows
```

Its shape is:

```
(2, 3)
```

That means:

```
2 rows
3 columns
```

---

4. Creating Your First NumPy Array

```
import numpy as np

numbers = np.array([10, 20, 30, 40])

print(numbers)
```

Output:

```
[10 20 30 40]
```

A two-dimensional array:

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

print(matrix)
```

Output:

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

NumPy arrays are generally homogeneous, meaning their elements have a common data type, and their shape is rectangular rather than jagged. These restrictions allow NumPy to optimize numerical operations.

---

5. Dimensions

NumPy arrays can have different numbers of dimensions.

1D Array

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

Shape:

```
(3,)
```

Think:

```
[1 2 3]
```

2D Array

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

Shape:

```
(3, 2)
```

Think:

```
1 2
3 4
5 6
```

3D Array

```
c = np.array([
   [
      [1, 2],
      [3, 4]
   ],
   [
      [5, 6],
      [7, 8]
   ]
])
```

Shape:

```
(2, 2, 2)
```

You can think of this as:

```
2 layers
  ×
2 rows
  ×
2 columns
```

NumPy's `ndarray` generalizes the array concept to an arbitrary number of dimensions.

---

6. Important Array Attributes

NumPy arrays contain useful attributes.

```
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])
```

ndim

Returns the number of dimensions:

```
print(data.ndim)
```

Output:

```
2
```

---

shape

Returns the size of each dimension:

```
print(data.shape)
```

Output:

```
(2, 3)
```

---

size

Returns the total number of elements:

```
print(data.size)
```

Output:

```
6
```

---

dtype

Returns the element data type:

```
print(data.dtype)
```

For example:

```
int64
```

These four attributes are fundamental when working with NumPy arrays.

---

7. Array Creation Functions

You do not always need to manually provide every value.

NumPy provides functions for generating arrays.

np.zeros()

```
zeros = np.zeros(5)

print(zeros)
```

Output:

```
[0. 0. 0. 0. 0.]
```

Create a matrix:

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

---

8. np.ones()

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

Output:

```
[1. 1. 1. 1. 1.]
```

2D:

```
ones = np.ones((3, 4))
```

You can explicitly choose the data type:

```
ones = np.ones(5, dtype=np.int64)
```

NumPy documents `zeros`, `ones`, `empty`, `arange`, and `linspace` as fundamental array-creation tools.

---

9. np.arange()

`arange()` creates values within a numerical range.

```
numbers = np.arange(0, 10)

print(numbers)
```

Output:

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

With a step:

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

Output:

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

The stop value is normally excluded.

---

10. np.linspace()

linspace() creates evenly spaced values across an interval.

```
numbers = np.linspace(0, 10, 5)

print(numbers)
```

Output:

```
[ 0.   2.5  5.   7.5 10. ]
```

Unlike arange(), linspace() is particularly useful when you care about how many values you want rather than the step size.

---

11. Choosing Between arange() and linspace()

Use:

```
np.arange(start, stop, step)
```

when you care about the **step**.

Use:

```
np.linspace(start, stop, count)
```

when you care about the **number of values**.

Example:

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

means:

`start = 0`
`stop = 10`
`step = 2`

Whereas:

`np.linspace(0, 10, 5)`

means:

`start = 0`
`stop = 10`
`values = 5`

---

12. Data Types

NumPy arrays have a specific `dtype`.

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

print(numbers.dtype)
```

You can explicitly specify one:

```
numbers = np.array(
    [1, 2, 3],
    dtype=np.float64
)
```

Now:

```
print(numbers)
```

produces values such as:

```
[1. 2. 3.]
```

Common NumPy types include:

```
np.int32
np.int64
np.float32
np.float64
np.bool_
np.complex64
np.complex128
```

Choosing an appropriate data type can matter when working with large datasets because data type affects memory usage and numerical behavior.

---

13. Indexing

NumPy indexing starts at zero, just like Python lists.

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

print(numbers[0])
```

Output:

```
10
```

```
print(numbers[2])
```

Output:

```
30
```

Negative indexing also works:

```
print(numbers[-1])
```

Output:

```
40
```

---

14. 2D Indexing

Consider:

```
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])
```

Access the first row:

```
data[0]
```

Output:

```
[10 20 30]
```

Access the second row:

```
data[1]
```

Access a specific element:

```
data[1, 2]
```

Output:

```
60
```

The general pattern is:

```
array[row, column]
```

---

15. Slicing

NumPy supports slicing.

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

print(numbers[1:4])
```

Output:

```
[20 30 40]
```

You can also use:

```
numbers[:3]
```

or:

```
numbers[2:]
```

or:

```
numbers[::2]
```

NumPy supports integer indexing, slicing, and more advanced indexing techniques.

---

16. 2D Slicing

```
data = np.array([
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
])
```

First two rows:

```
data[:2]
```

First two columns:

```
data[:, :2]
```

The comma separates dimensions:

```
rows, columns
```

So:

```
data[1:, 1:]
```

means:

```
rows from index 1 onward
columns from index 1 onward
```

---

17. Changing Array Values

NumPy arrays are mutable.

```
numbers = np.array([10, 20, 30])

numbers[1] = 99

print(numbers)
```

Output:

```
[10 99 30]
```

For a 2D array:

```
data[0, 1] = 100
```

---

18. Vectorized Arithmetic

This is one of NumPy's most important ideas.

Consider:

```
data = np.array([10, 20, 30])
```

Add 5:

```
data + 5
```

Result:

```
[15 25 35]
```

Multiply by 2:

```
data * 2
```

Result:

```
[20 40 60]
```

Divide:

```
data / 10
```

Result:

```
[1. 2. 3.]
```

NumPy performs these operations element by element without requiring you to write a Python loop.

---

19. Array-to-Array Operations

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

Addition:

```
a + b
```

Result:

```
[11 22 33]
```

Subtraction:

```
a - b
```

Result:

```
[9 18 27]
```

Multiplication:

```
a * b
```

Result:

```
[10 40 90]
```

Division:

```
a / b
```

Result:

```
[10. 10. 10.]
```

For element-wise operations, compatible shapes are required.

---

20. Element-Wise Operations vs Matrix Multiplication

This distinction is extremely important.

Given:

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

This:

```
a * a
```

performs element-wise multiplication:

```
[[1, 4],
 [9, 16]]
```

It is **not** matrix multiplication.

Matrix multiplication uses:

```
a @ a
```

Result:

```
[[ 7 10]
 [15 22]]
```

This distinction becomes especially important in data science, machine learning, and linear algebra.

---

21. Aggregation Functions

NumPy provides functions for summarizing numerical data.

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

Sum:

```
data.sum()
```

Result:

```
100
```

Minimum:

```
data.min()
```

Maximum:

```
data.max()
```

Average:

```
data.mean()
```

Standard deviation:

```
data.std()
```

Product:

```
data.prod()
```

NumPy provides common aggregation operations such as sum, minimum, maximum, mean, product, and standard deviation.

---

22. The `axis` Concept

`axis` is one of the most confusing NumPy concepts for beginners.

Consider:

```
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])
```

Shape:

```
(2, 3)
```

There are:

```
2 rows
3 columns
```

Now:

```
data.sum()
```

gives:

```
210
```

But:

```
data.sum(axis=0)
```

gives:

```
[50 70 90]
```

This combines values vertically:

```
10 + 40 = 50
20 + 50 = 70
30 + 60 = 90
```

---

23. axis=1

Now:

```
data.sum(axis=1)
```

gives:

```
[60 150]
```

Because:

```
10 + 20 + 30 = 60
40 + 50 + 60 = 150
```

A useful mental model:

```
axis=0
↓
collapse rows
→ result for each column

axis=1
→
collapse columns
↓ result for each row
```

NumPy's documentation demonstrates this distinction with row and column aggregations.

---

24. Reshaping Arrays

Sometimes the data is correct but arranged in the wrong shape.

```
numbers = np.arange(1, 7)

print(numbers)
```

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

Reshape it:

```
matrix = numbers.reshape(2, 3)
```

Result:

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

The number of elements must remain the same.

Original:

```
6 elements
```

New:

```
2 × 3 = 6
```

---

25. Flattening

You can convert a multidimensional array into one dimension.

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

flat = data.flatten()
```

Result:

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

Another commonly used operation is:

```
data.ravel()
```

For beginners, remember:

reshape → change organization

flatten/ravel → turn dimensions into one dimension

---

26. Transpose

Transpose switches rows and columns.

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

Use:

```
data.T
```

Result:

```
[[1 4]
 [2 5]
```

```
 [3 6]]
```

The original shape:

```
(2, 3)
```

becomes:

```
(3, 2)
```

---

27. Combining Arrays

You can combine arrays using functions such as `concatenate`.

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

result = np.concatenate((a, b))
```

Result:

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

NumPy also provides stacking operations such as:

```
np.vstack()
np.hstack()
```

These are useful when building larger arrays from smaller ones.

---

28. Boolean Conditions

NumPy allows comparisons across an entire array.

```
scores = np.array([45, 67, 82, 91, 38])

scores > 60
```

Result:

```
[False  True  True  True False]
```

This produces a Boolean array.

You can use it to select values.

```
scores[scores > 60]
```

Result:

```
[67 82 91]
```

This is called **Boolean indexing** or **Boolean masking**.

---

29. Boolean Filtering

Example:

```
temperatures = np.array([
   18, 22, 27, 31, 35, 20
```

```
])

hot = temperatures[temperatures > 30]

print(hot)
```

Output:

```
[31 35]
```

This is extremely useful in data processing.

---

30. Multiple Conditions

Use parentheses around individual conditions.

```
scores = np.array([45, 67, 82, 91, 38])

result = scores[
    (scores >= 60) & (scores <= 90)
]
```

Result:

```
[67 82]
```

For NumPy arrays, use:

```
&  → AND
|  → OR
~  → NOT
```

rather than Python's `and`, `or`, and `not` for element-wise Boolean conditions.

---

31. Universal Functions

NumPy provides **universal functions**, commonly called **ufuncs**, which operate element by element on arrays and support broadcasting.

For example:

```
numbers = np.array([1, 4, 9, 16])

np.sqrt(numbers)
```

Result:

```
[1. 2. 3. 4.]
```

Other examples:

```
np.abs(data)
np.exp(data)
np.log(data)
np.sin(data)
np.cos(data)
np.round(data)
```

This lets you apply mathematical functions to entire arrays.

---

32. Broadcasting

Broadcasting allows NumPy to perform operations between arrays with compatible shapes.

For example:

```
data = np.array([10, 20, 30])

data + 5
```

NumPy conceptually treats the scalar as though it were:

```
[5, 5, 5]
```

and produces:

```
[15, 25, 35]
```

It does not necessarily create those repeated values in memory just to perform the operation.

---

33. Broadcasting With 2D Arrays

Consider:

```
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])

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

data + bonus
```

Result:

```
[[11 22 33]
 [41 52 63]]
```

The one-dimensional array is broadcast across the rows.

Broadcasting compares dimensions from the rightmost side. Dimensions are compatible when they are equal or when one of them is `1`.

---

34. Broadcasting Errors

This will fail:

```
data = np.array([
    [10, 20, 30],
    [40, 50, 60]
])

values = np.array([1, 2])

data + values
```

Why?

Shapes:

```
data   → (2, 3)
values → (2,)
```

Compare from the right:

`3 vs 2`

Neither is equal and neither is `1`.

Therefore, the shapes are incompatible.

NumPy raises a `ValueError` when broadcasting rules cannot make the shapes compatible.

---

35. `np.newaxis`

You can add a dimension with:

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

values[:, np.newaxis]
```

Now the shape changes from:

`(3,)`

to:

`(3, 1)`

This can make broadcasting possible.

For example:

```
a = np.array([1, 2, 3, 4])[:, np.newaxis]
b = np.array([10, 20, 30])
```

```
result = a + b
```

Result:

```
[[11 21 31]
 [12 22 32]
 [13 23 33]
 [14 24 34]]
```

The NumPy documentation uses `newaxis` as a way to insert a dimension for broadcasting.

---

36. Copy vs View

This is an important concept when modifying arrays.

Sometimes an operation gives you a view into existing data rather than an independent copy.

For example:

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

view = data[1:3]

view[0] = 99
```

Now:

```
print(data)
```

may show:

```
[10 99 30 40]
```

because the slice can refer to the same underlying data.

If you need an independent array:

```
copy = data[1:3].copy()
```

Then changes to `copy` do not modify the original array.

Understanding copies and views is important for both correctness and memory efficiency.

---

37. Sorting

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

sorted_numbers = np.sort(numbers)

print(sorted_numbers)
```

Output:

```
[10 20 30 40]
```

`np.sort()` returns sorted data.

NumPy also provides related tools such as `argsort`, `searchsorted`, and `partition`.

---

38. Finding Minimum and Maximum Positions

Suppose:

```
scores = np.array([75, 91, 64, 88])
```

Find the position of the smallest value:

```
np.argmin(scores)
```

Find the position of the largest:

```
np.argmax(scores)
```

These functions return indexes rather than the values themselves.

---

39. Random Numbers

NumPy provides random-number functionality.

Modern NumPy code commonly uses a random generator:

```
rng = np.random.default_rng()

numbers = rng.integers(
    1,
    100,
    size=5
)

print(numbers)
```

You can also generate random floating-point values:

```
numbers = rng.random(5)
```

Random arrays are useful for:

```
simulations
testing
experiments
sampling
machine learning
data generation
```

For reproducible experiments, provide a seed:

```
rng = np.random.default_rng(42)
```

Then the same generator configuration can reproduce the same sequence.

---

40. A Practical Data Analysis Example

Imagine a developer has response-time measurements:

```
response_times = np.array([
    120,
    145,
    98,
    210,
    180,
    130,
    160
])
```

Average:

```
response_times.mean()
```

Fastest:

```
response_times.min()
```

Slowest:

```
response_times.max()
```

Standard deviation:

```
response_times.std()
```

Requests above 150 ms:

```
slow = response_times[response_times > 150]
```

Now the same data can be analyzed without manually looping through every value.

---

41. NumPy With Real-World Data

NumPy commonly becomes part of a larger data workflow:

```
CSV / Database / API
        ↓
    NumPy array
        ↓
  Clean / transform
        ↓
   Calculate
```

Analyze

Visualization / ML / Report

NumPy is therefore often a foundation rather than the final application.

It works especially well alongside tools such as pandas, Matplotlib, and machine-learning libraries.

---

42. NumPy vs Python Lists

Feature	Python List	NumPy Array
General-purpose	Yes	Mainly numerical
Mixed data types	Yes	Generally homogeneous
Multidimensional data	Possible but awkward	Native
Vectorized arithmetic	No	Yes
Broadcasting	No	Yes
Numerical functions	Limited	Extensive

Numerical performance	General purpose	Optimized for array operations
Flexible element types	Very flexible	More constrained
Common use	General Python	Data/science/ML/numerical computing

NumPy is not a replacement for Python lists everywhere.

Use lists when you need a general-purpose Python collection.

Use NumPy when your problem is primarily numerical array computation.

---

43. NumPy vs Loops

Python loop:

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

result = []

for number in numbers:
    result.append(number * 10)
```

NumPy:

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

```
result = numbers * 10
```

The NumPy version communicates the numerical operation directly.

NumPy's vectorized operations and ufuncs can perform array operations efficiently, with much of the computation implemented outside ordinary Python-level loops.

However, do not assume that every NumPy expression is automatically faster. Poorly designed operations, unnecessary copies, or excessive temporary arrays can still waste memory or time.

---

44. A Complete NumPy Workflow

A typical workflow looks like:

```
Create / load data
    ↓
Inspect shape and dtype
    ↓
Select data
    ↓
Transform data
    ↓
Perform calculations
    ↓
Aggregate results
    ↓
Export / visualize / use elsewhere
```

For example:

```
import numpy as np
```

```
scores = np.array([
    72,
    85,
    91,
    64,
    88
])

print("Shape:", scores.shape)
print("Average:", scores.mean())
print("Highest:", scores.max())
print("Lowest:", scores.min())

passing = scores[scores >= 70]

print("Passing:", passing)
```

Output conceptually:

```
Shape: (5,)
Average: 80.0
Highest: 91
Lowest: 64
Passing: [72 85 91 88]
```

---

45. Common NumPy Mistakes

Mistake 1: Forgetting the shape

Before operating on arrays, inspect:

`array.shape`

Shape explains how NumPy will interpret the data.

Mistake 2: Confusing `*` with `@`

`A * B`

means element-wise multiplication.

`A @ B`

means matrix multiplication.

Mistake 3: Using Python `and` / `or`

Avoid:

`scores > 50 and scores < 90`

Use:

`(scores > 50) & (scores < 90)`

Mistake 4: Ignoring broadcasting

Always check shapes when an operation involves different-sized arrays.

```
print(a.shape)
print(b.shape)
```

Mistake 5: Accidentally modifying shared data

Be aware of views.

Use:

```
.copy()
```

when you need independent data.

Mistake 6: Assuming every operation is faster

NumPy is optimized for array operations, but unnecessary intermediate arrays and poorly chosen operations can still hurt performance.

Mistake 7: Treating NumPy arrays exactly like lists

For example, NumPy's arithmetic semantics differ from Python lists:

```
[1, 2, 3] * 2
```

produces:

```
[1, 2, 3, 1, 2, 3]
```

whereas:

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

produces:

```
[2 4 6]
```

Know which data structure you are using.

---

46. Best Practices

1. Inspect your arrays

Use:

```
array.shape
array.ndim
array.size
array.dtype
```

2. Prefer vectorized operations

Instead of manually looping through numerical data when NumPy already provides an array operation, use the NumPy operation.

3. Think in shapes

Ask:

```
What is the shape?
What shape do I want?
Are these shapes compatible?
```

4. Be intentional with data types

For large datasets, dtype can affect memory consumption and numerical precision.

5. Understand views and copies

Do not assume every derived array is independent.

6. Use clear variable names

Prefer:

`temperature_data`

over:

`x`

when the meaning matters.

7. Start with small arrays

When learning a new NumPy operation, inspect a small example first.

---

47. Mini Project: Student Score Analyzer

Build a program that analyzes student scores.

Input:

`scores = np.array([`

```
    72, 85, 91, 64, 88,
    76, 95, 58, 81, 90
])
```

The program should calculate:

```
Average score
Highest score
Lowest score
Standard deviation
Number of passing students
Scores above 90
```

Useful NumPy operations:

```
scores.mean()
scores.max()
scores.min()
scores.std()
scores[scores >= 60]
scores[scores > 90]
```

Then calculate the number of passing students:

```
passing = scores[scores >= 60]

print("Passing students:", passing.size)
```

---

48. 5-Minute Challenge

Create:

```
temperatures = np.array([
    22, 25, 27, 31, 29,
    35, 33, 26, 24, 30
])
```

Without writing a loop, calculate:

1. Average temperature
2. Highest temperature
3. Lowest temperature
4. Temperatures above 30
5. Number of temperatures above 30
6. Difference between the highest and lowest temperature

---

49. Exercises

Exercise 1

Create a 1D array containing numbers from 1 to 20.

Exercise 2

Create a `3 × 4` array containing zeros.

Exercise 3

Create a `5 × 5` array containing ones.

Exercise 4

Create ten evenly spaced values between 0 and 1.

Exercise 5

Create a 3×3 matrix and print:

- its shape
- number of dimensions
- number of elements
- data type

Exercise 6

Extract the second row of a 2D array.

Exercise 7

Extract the last two columns.

Exercise 8

Find all values greater than 50.

Exercise 9

Calculate row-wise and column-wise sums.

Exercise 10

Reshape an array containing 12 values into:

2×6
3×4
4×3
6×2

Exercise 11

Transpose a `2 × 4` array.

Exercise 12

Create two arrays and experiment with broadcasting.

---

50. Mini Quiz

1. What is NumPy mainly designed for?

A. Web routing
B. Numerical and array-based computation
C. HTML rendering
D. Database authentication

Answer: B

2. What is NumPy's main array object called?

A. `list`
B. `table`
C. `ndarray`
D. `matrixlist`

Answer: C

3. What does `shape` tell you?

A. The total memory used
B. The dimensions and size along each axis
C. The largest value
D. The data type

Answer: B

4. What does `axis=0` commonly mean for a 2D array aggregation?

A. Aggregate across each column
B. Aggregate across each row
C. Delete the first row
D. Reverse the array

Answer: A

5. What does broadcasting allow?

A. Sending data to a server
B. Combining arrays with compatible different shapes
C. Converting Python into JavaScript
D. Saving arrays as images

Answer: B

6. What is the difference between `*` and `@`?

A. They are identical
B. `*` is element-wise multiplication; `@` is matrix multiplication
C. `*` is matrix multiplication; `@` is division
D. `@` is only used with lists

Answer: B

---

51. Interview Questions

Beginner

1. What is NumPy?
2. Why use NumPy instead of Python lists for numerical data?
3. What is an `ndarray`?
4. What does `shape` mean?
5. What is `dtype`?
6. What does `ndim` represent?
7. What does `size` represent?

Intermediate

8. What is vectorization?
9. What is broadcasting?
10. What is the difference between `axis=0` and `axis=1`?
11. What is Boolean indexing?
12. What is the difference between `reshape()` and `flatten()`?
13. What is the difference between a view and a copy?
14. What is the difference between `*` and `@`?
15. When would you use `arange()` versus `linspace()`?

Advanced

16. Explain NumPy broadcasting rules.
17. Why are homogeneous arrays useful for numerical computing?
18. How can dtype affect memory usage?
19. Why can unnecessary intermediate arrays hurt performance?

20. What are ufuncs?
21. How would you debug a broadcasting error?
22. Why is vectorized NumPy code often more efficient than equivalent Python-level loops?

---

52. NumPy Cheat Sheet

Import

```
import numpy as np
```

Create array

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

Zeros

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

Ones

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

Range

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

Evenly spaced values

```
np.linspace(0, 10, 5)
```

Shape

```
arr.shape
```

Dimensions

`arr.ndim`

Number of elements

`arr.size`

Data type

`arr.dtype`

Reshape

`arr.reshape(2, 3)`

Transpose

`arr.T`

Sum

`arr.sum()`

Mean

`arr.mean()`

Minimum

`arr.min()`

Maximum

`arr.max()`

Standard deviation

`arr.std()`

Sort

`np.sort(arr)`

Filter

`arr[arr > 50]`

Square root

`np.sqrt(arr)`

Absolute value

`np.abs(arr)`

Concatenate

`np.concatenate((a, b))`

Matrix multiplication

`a @ b`

Copy

`arr.copy()`

---

53. The NumPy Mental Model

When you see NumPy code, think in this order:

1. What data do I have?

2. What is its shape?

3. What is its dtype?

4. Which elements do I need?

5. What transformation do I want?

6. Can NumPy perform it vectorially?

7. What axis should the operation use?

8. Are the shapes broadcast-compatible?

9. Do I need a view or a copy?

10. What should the resulting shape be?

This way of thinking is more important than memorizing individual NumPy functions.

---

54. Key Takeaways

1. NumPy is a foundational Python library for numerical computing.
2. Its central object is the multidimensional `ndarray`.
3. NumPy arrays have a shape, dimensions, size, and data type.
4. Arrays can be created with functions such as `array`, `zeros`, `ones`, `arange`, and `linspace`.
5. NumPy supports powerful indexing and slicing.

6. Vectorized operations allow numerical calculations across entire arrays.
7. Aggregation functions summarize numerical data.
8. `axis` determines the dimension along which many operations are performed.
9. Broadcasting allows operations between compatible shapes.
10. Boolean indexing provides powerful filtering.
11. `*` performs element-wise multiplication, while `@` performs matrix multiplication.
12. Views and copies matter when modifying array data.
13. NumPy provides a foundation for tools used in data analysis, scientific computing, visualization, and machine learning.
14. The key NumPy skill is learning to think in **arrays, shapes, axes, and transformations**.

References

NumPy Documentation: https://numpy.org/doc/stable/

NumPy Absolute Beginner's Guide: https://numpy.org/doc/stable/user/absolute_beginners.html

NumPy Quickstart: https://numpy.org/doc/stable/user/quickstart.html

NumPy Broadcasting: https://numpy.org/doc/stable/user/basics.broadcasting.html

NumPy User Guide: https://numpy.org/doc/stable/user/

Visit haas.dev for more resources and guides.

Website: https://dev-roast-app.vercel.app

Resources: https://dev-roast-app.vercel.app/resources