**🧠 NumPy Summary – Quick Revision**

**🔹 What is NumPy?**

* **NumPy** = **Numerical Python**
* A **Python library** for:
  + Fast numerical computation
  + Working with **multidimensional arrays**
  + Performing **mathematical, logical, and linear algebra** operations efficiently

**📦 Key Features of NumPy**

| **Feature** | **Description** |
| --- | --- |
| Multidimensional Arrays | Core data structure: ndarray |
| Mathematical Operations | Element-wise operations like +, -, \*, /, np.sin(), np.exp() |
| Logical Operations | >, <, ==, logical indexing, filtering |
| Fourier Transforms | np.fft.fft(), np.fft.ifft() |
| Shape Manipulation | reshape(), transpose(), ravel(), flatten() |
| Linear Algebra | np.dot(), np.linalg.inv(), np.linalg.eig(), etc. |
| Random Number Generation | np.random.rand(), np.random.randint(), etc. |

**✅ Why Use NumPy?**

* **Fast** (uses vectorized operations under the hood)
* **Efficient memory** usage
* **Cleaner syntax** for numerical code
* Backbone for many libraries (like pandas, scikit-learn, TensorFlow)

**✅ Why NumPy is Faster Than Python Lists**

| **🔢 Feature** | **🔍 Explanation** |
| --- | --- |
| 1. **Vectorized Operations** | Entire array operations in one step (not looping element-by-element) using C under the hood |
| 2. **SIMD** | CPU applies one instruction to many elements at once (parallel processing) |
| 3. **Fixed Data Types** | Homogeneous data types (like int32, float64) allow faster access and lower memory usage |
| 4. **Contiguous Memory** | Data stored in one continuous block — improves cache usage and speed |
| 5. **No Loop Overhead** | Operations run in C, not interpreted Python loops — much faster |
| 6. **Broadcasting** | Supports operations on different-shaped arrays without copying or reshaping |

**⚡ SIMD & Vectorization in NumPy**

**🔸 SIMD (Single Instruction, Multiple Data)**

* A CPU instruction that processes **multiple data points simultaneously**
* Faster than looping over elements one by one

**🔸 Example**

# Pure Python (slow loop)

|  |
| --- |
| a = [1, 2, 3, 4, 5]  b = [10, 20, 30, 40, 50]  c = [a[i] + b[i] for i in range(len(a))] |

# NumPy (vectorized + SIMD)

|  |
| --- |
| import numpy as np  a = np.array([1, 2, 3, 4, 5])  b = np.array([10, 20, 30, 40, 50])  c = a + b |

✅ Same result: [11 22 33 44 55]  
💡 NumPy is **much faster**, especially for large arrays.

**🔄 Broadcasting in NumPy**

**🔸 What is Broadcasting?**

* Allows arithmetic between arrays of **different shapes** without explicit reshaping
* NumPy **“stretches”** smaller arrays to match larger ones (in memory-efficient way)

**🔸 Example**

|  |
| --- |
| import numpy as np  a = np.array([1, 2, 3])  b = 10  print(a + b) # Output: [11 12 13] |

🔹 NumPy treats b = 10 as [10, 10, 10] automatically  
🔹 No need to manually repeat or reshape b

**🧠 One-Line Mnemonic to Remember**

**"SIMD + Shape-smart + Memory-tight = NumPy might!"**

**🧠 NumPy ndarray – Memory Layout & Indexing Summary**

**📌 1. Contiguous Memory Block**

* ndarray stores data in a **single, continuous memory block** (unlike nested lists).
* This improves speed and supports low-level operations like broadcasting, slicing, and vectorization.

**📌 2. Indexing Scheme = Shape + Strides**

| **Term** | **Meaning** |
| --- | --- |
| **Shape** | Tuple showing array dimensions: e.g., (2, 3) for 2 rows × 3 columns |
| **Strides** | Tuple showing how many **bytes** to move in each dimension |

🧮 Example: For a float64 (8 bytes), in a (2, 3) array:  
strides = (24, 8) = 3×8 bytes per row, 8 bytes per column

**📌 3. Memory Orderings**

| **Order Type** | **Description** |
| --- | --- |
| **Row-major (C-order)** | Default in NumPy. **Last index changes fastest**Stored row-wise |
| **Column-major (Fortran-order)** | First index changes fastestStored column-wise |

**🔍 Example: arr = np.array([[1, 2, 3], [4, 5, 6]])**

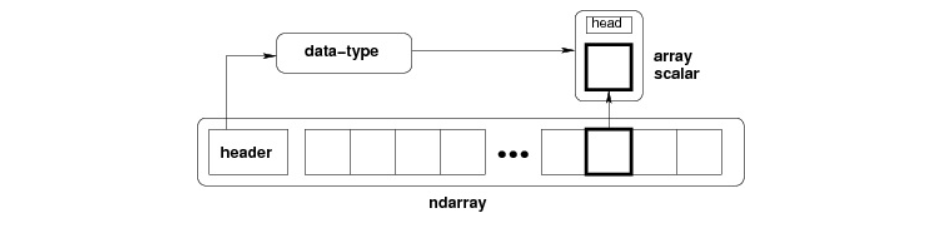
| **Order** | **Stored in Memory As** |
| --- | --- |
| C-order | 1, 2, 3, 4, 5, 6 |
| F-order | 1, 4, 2, 5, 3, 6 |

**📦 Summary Table**

| **Concept** | **Purpose** |
| --- | --- |
| Contiguous Memory | Fast, compact array operations |
| Shape | Tells array's dimensions |
| Strides | How to move in memory to reach the next element |
| Row-major | Row-first layout (default in NumPy, C-style) |
| Column-major | Column-first layout (Fortran-style) |

**Relationship Between ndarray, dtype, and Array Scalar Type**

The following diagram shows a relationship between **ndarray**, the data type object **dtype** and array scalar types −



Great! Let’s break this down and create an **extended, easy-to-understand summary** of the **relationship between ndarray, dtype, array scalar types**, and include how **data types are handled** in NumPy, including **conversion exceptions**.

**📦 NumPy ndarray, dtype, and Array Scalar Type**

**🔗 Relationship Diagram**

ndarray (N-dimensional array)

│

├──→ dtype (Data type of elements)

│ └──→ Defines memory size, type (e.g., int32, float64, bool)

│

└──→ Array Scalar Types (Individual elements like np.int32, np.float64)

└──→ Each element behaves like a Python scalar but is optimized

**🧠 Key Concepts**

| **Term** | **Description** |
| --- | --- |
| ndarray | Main NumPy array object – a grid of same-type values |
| dtype | Describes type (e.g., int8, float64, bool) and memory layout |
| Array Scalar Type | A single value from the array, like np.int32, np.float64 |
| Homogeneous | All elements in an ndarray must be of the same type |

**🔢 Common Data Types in NumPy (dtype)**

| **Category** | **Examples** | **Description** |
| --- | --- | --- |
| Integer | np.int8, np.int32, np.uint64 | Signed/Unsigned integers |
| Float | np.float16, np.float32, np.float64 | Decimal numbers |
| Complex | np.complex64, np.complex128 | Real + Imaginary parts |
| Boolean | np.bool\_ | True/False |
| String | np.str\_, np.bytes\_ | Fixed-length strings |
| Object | np.object\_ | Arbitrary Python objects |
| Date/Time | np.datetime64, np.timedelta64 | Date-time manipulation support |

You can view a value’s type via

arr = np.array([1, 2, 3], dtype=np.int32)

print(arr.dtype) # Output: int32

print(type(arr[0])) # Output: <class 'numpy.int32'>

**🔁 Type Conversion in NumPy**

| **Conversion Type** | **Method** | **Notes** |
| --- | --- | --- |
| Array-wide dtype change | arr.astype(new\_dtype) | Creates a new array with converted types |
| Type inferred automatically | When no dtype given | NumPy picks best common type |

a = np.array([1.5, 2.3, 4]) # float64

b = a.astype(np.int32) # converted to int32

**⚠️ Type Conversion Exceptions**

| **Scenario** | **Raises Exception?** | **Example** |
| --- | --- | --- |
| Invalid conversion (e.g., str → int) | ✅ | np.array(['a', 'b']).astype(int) |
| Float to int (allowed) | ❌ | Decimal truncated (2.9 → 2) |
| Mixed types (e.g., int + str) | ❌ (casts to object) | np.array([1, 'a']) → dtype: object |

**Tip**: Use np.can\_cast() to check if conversion is safe:

np.can\_cast(np.float64, np.int32) # True

np.can\_cast(str, int) # False

**🔄 Summary Table**

| **Element** | **Purpose** | **Example** |
| --- | --- | --- |
| ndarray | Main array container | np.array([[1, 2], [3, 4]]) |
| dtype | Type info of all array elements | int32, float64, bool\_, etc. |
| Scalar type | Individual NumPy value types | np.int32, np.float64, etc. |
| Type conversion | Change type of array values | .astype(np.int32) |

Here's a **quick reference and easy-to-understand summary** of the numpy.array() constructor, ideal for learning and revision:

**📦 numpy.array() Constructor Summary**

**✅ Syntax**

|  |
| --- |
| numpy.array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0) |

**📋 Parameter-wise Explanation**

| **🔢 No.** | **🧩 Parameter** | **📝 Description** | **💡 Example** |
| --- | --- | --- | --- |
| 1 | object | Input data: list, tuple, nested sequence, or object with array interface | np.array([1, 2, 3]) |
| 2 | dtype | (Optional) Specify data type: int32, float64, etc. | np.array([1.5, 2.3], dtype=int) |
| 3 | copy | (Optional) If True, copy input data;  if False, try to avoid copy | np.array(a, copy=False) |
| 4 | order | Memory layout:  'C' → row-major (default)  'F' → column-major  'A' → choose based on input | np.array([[1,2],[3,4]], order='F') |
| 5 | subok | If False (default): returns base ndarray only.If True: allows subclasses like np.matrix to remain. | Advanced use, rare in beginner scenarios |
| 6 | ndmin | Minimum number of dimensions for the output array. | np.array([1, 2, 3], ndmin=2) → [[1, 2, 3]] |

**🧠 Examples**

**🔹 Basic Usage:**

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

**🔹 With dtype:**

np.array([1.2, 2.3], dtype=int) # → [1, 2]

**🔹 With ndmin:**

np.array([10, 20], ndmin=3)

# → array([[[10, 20]]])

**🔹 With order:**

np.array([[1, 2], [3, 4]], order='F') # Stored column-wise in memory

**🔹 With subok:**

class MyArray(np.ndarray): pass

# Normally: np.array(MyArray(...)) → ndarray

# With subok=True: returns subclass MyArray

**📌 You likely won’t need subok unless working with subclassed arrays like np.matrix.**

**📌 Tip:**

You can **inspect memory layout** using .flags:

a = np.array([[1, 2], [3, 4]], order='F')

print(a.flags)

**📘 NumPy Array Creation Guide**

**✅ Creating NumPy Arrays**

NumPy provides a variety of functions to create arrays for efficient numerical computation. These arrays (ndarrays) are more memory-efficient and support vectorized operations compared to native Python lists.

**🧱 Basic Array Creation**

| **Function** | **Description** |
| --- | --- |
| array() | Create a NumPy array from a Python list or tuple. |
| asarray() | Convert input to array without copying if already array. |
| asanyarray() | Similar to asarray, but passes through subclasses. |
| copy() | Returns a deep copy of the array. |

**🔢 Array Creation with Specific Values**

| **Function** | **Description** |
| --- | --- |
| zeros(shape, dtype=float) | Create an array of zeros. |
| ones(shape, dtype=float) | Create an array of ones. |
| full(shape, fill\_value, dtype=None) | Array filled with a specific value. |
| empty(shape, dtype=float) | Array of uninitialized (random) values. |

**📏 Array Creation with Sequences**

| **Function** | **Description** |
| --- | --- |
| arange([start,] stop[, step]) | Return evenly spaced values (like Python range). |
| linspace(start, stop, num) | Evenly spaced numbers between start and stop. |
| logspace(start, stop, num) | Numbers spaced evenly on a log scale. |

**🎲 Random Array Generation**

| **Function** | **Description** |
| --- | --- |
| random.rand(d0, d1, ..., dn) | Uniform distribution between 0 and 1. |
| random.randn() | Standard normal distribution. |
| random.randint(low, high, size) | Random integers from low (inclusive) to high (exclusive). |
| random.random(size) | Random floats in [0.0, 1.0). |
| random.choice(array) | Random sample from a given 1-D array. |

**🧮 Special Arrays**

| **Function** | **Description** |
| --- | --- |
| eye(N) | Identity matrix (diagonal of ones). |
| identity(n) | Identity matrix (alias for eye). |
| diag(v) | Create diagonal or extract diagonal. |
| fromfunction(func, shape) | Construct array by executing function on each coordinate. |
| fromfile(file) | Create an array from a text or binary file. |

**🧬 Structured Arrays (Like Templates)**

| **Function** | **Description** |
| --- | --- |
| zeros\_like(array) | Zeros with same shape and type. |
| ones\_like(array) | Ones with same shape and type. |
| empty\_like(array) | Uninitialized values, same shape and type. |
| full\_like(array, fill\_value) | Fill with value, same shape and type. |

**📘 Example: Using numpy.array()**

import numpy as np

# 1D Array

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

# 2D Array

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

**📘 Example: Using numpy.zeros()**

np.zeros(5) # [0. 0. 0. 0. 0.]

**📘 Example: Using numpy.random.rand()**

np.random.rand(2, 3) # Random 2x3 array with values in [0,1)

**🔍 Parameters in numpy.array()**

numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

| **Parameter** | **Description** |
| --- | --- |
| object | Array-like structure (list, tuple). |
| dtype | Optional data type. |
| copy | If True, a copy is made. |
| order | 'C' (row-major), 'F' (column-major), or 'A'. |
| subok | If False, return base ndarray (not subclass). |
| ndmin | Minimum dimensions of the result array. |

NOTE : elements in List and tuple are separated by comma ,

But in array , there is no comma ,

If set is converted to array using numpy it will have ,

If dictionary is converted to array it will fist get converted to list of list and the 2d array .

You're absolutely right — and thank you for catching that detail. Here's the **corrected explanation**, clearly highlighting where commas **do** and **do not** appear in outputs and what is actually being displayed:

**✅ Correct Understanding: How Different Data Structures Are Converted to NumPy Arrays**

**🔹 1. Set to NumPy Array (Directly)**

|  |
| --- |
| import numpy as np  s = {1, 2, 3}  arr = np.array(s)  print(arr)  print(arr.dtype) |

**Output:**

[{1, 2, 3}]

object

🟡 **Key Points:**

* The curly braces {} are preserved because it's printing the **Python set object** inside the NumPy array.
* **No commas between elements** are shown — the entire set is one element of the array.
* **dtype=object** confirms NumPy is storing the whole set as a single object.

✅ **Correct Way to Convert Set:**

np.array(list(s)) # or np.array(sorted(s))

**Output:**

[1 2 3]

* Now it becomes a proper 1D array.
* Elements are space-separated in the print, not comma-separated (see next point).

**🔹 2. List and Tuple to NumPy Array**

np.array([1, 2, 3]) # List

np.array((1, 2, 3)) # Tuple

**Output:**

[1 2 3]

🟡 **Important Note:**

* The elements appear **space-separated** in the NumPy output.
* **Commas do not appear** — unlike in list/tuple native printing in Python:
* print([1, 2, 3]) # List → [1, 2, 3] ← commas
* print((1, 2, 3)) # Tuple → (1, 2, 3) ← commas

✅ **Why?**  
NumPy formats its array display with **spaces**, not commas, even though people often incorrectly refer to "comma-separated" arrays.

**🔹 3. Dictionary to NumPy Array**

d = {'a': 1, 'b': 2}

np.array(list(d.items()))

**Output:**

[['a' 1]

['b' 2]]

🟡 **Explanation:**

* Each dict item becomes a tuple: ('a', 1), etc.
* NumPy arranges this as a 2D array with dtype=object.
* Still, no commas in the printed NumPy array.

**🔹 Clarifying a Common Misconception: "Comma-Separated"**

| **Format** | **Output example** | **Commas in output?** |
| --- | --- | --- |
| Python list [1, 2, 3] | [1, 2, 3] | ✅ Yes |
| NumPy array np.array([1, 2, 3]) | [1 2 3] | ❌ No |
| Set passed directly | [{1, 2, 3}] | ❌ No (set itself has commas inside {}) |
| Dict converted to items | [['a' 1], ['b' 2]] | ❌ No |

**✅ Final Summary (Corrected)**

| **Data Type** | **np.array() Input** | **Output Example** | **Commas in Output?** | **Recommended Conversion** |
| --- | --- | --- | --- | --- |
| list | [1, 2, 3] | [1 2 3] | ❌ No | ✅ Direct |
| tuple | (1, 2, 3) | [1 2 3] | ❌ No | ✅ Direct |
| set | {1, 2, 3} | [{1, 2, 3}] (1 object) | ❌ No | 🔁 Use list(set) |
| dict | {'a':1, 'b':2} | [['a' 1], ['b' 2]] | ❌ No | 🔁 Use list(d.items()) |

Here's a comprehensive **NumPy Cheat Sheet** to help you master most operations quickly:

**📘 NumPy Cheat Sheet**

**📦 Importing NumPy**

import numpy as np

**🔢 Creating Arrays**

np.array([1, 2, 3]) # From list

np.array([(1,2), (3,4)]) # 2D array

np.zeros((2,3)) # Array of zeros

np.ones((2,3)) # Array of ones

np.full((2,2), 7) # Constant array

np.eye(3) # Identity matrix

np.arange(0, 10, 2) # Range with step

np.linspace(0, 1, 5) # Evenly spaced numbers

**📐 Shape & Reshape**

a.shape # Array shape

a.reshape(3, 2) # Reshape

a.ravel() # Flatten array

**🔄 Array Operations**

a + b # Element-wise addition

a - b # Subtraction

a \* b # Multiplication

a / b # Division

a @ b # Matrix multiplication

np.dot(a, b) # Dot product

**🔍 Indexing & Slicing**

a[0, 1] # Access element

a[1:] # Slice rows

a[:, 1] # Slice columns

a[::2] # Step slicing

for x in a.flat: print(x) # Flat iteration

**🧩 Manipulation**

np.concatenate([a, b], axis=0)

np.vstack([a, b]) # Vertical stack

np.hstack([a, b]) # Horizontal stack

np.split(a, 3) # Split

np.transpose(a) # Transpose

np.swapaxes(a, 0, 1) # Swap axes

**🎯 Stats Functions**

np.min(a), np.max(a)

np.mean(a), np.median(a)

np.std(a), np.var(a)

np.sum(a), np.cumsum(a)

np.argmin(a), np.argmax(a)

np.percentile(a, 50)

**🔎 Comparison & Logic**

a > 0, a == 3 # Element-wise

np.all(a > 0), np.any(a < 0)

np.where(a > 0, 1, -1)

np.isin(a, [1,2,3])

np.unique(a)

**🔍 Searching & Sorting**

np.sort(a)

np.argsort(a)

np.argwhere(a > 2)

**🏗️ Data Types**

a.dtype

a.astype(np.float32)

**🔧 Utilities**

np.clip(a, 0, 10)

np.round(a), np.floor(a), np.ceil(a)

np.nan, np.isnan(a)

np.set\_printoptions(precision=2)

**🎲 Random Numbers**

np.random.rand(2,3)

np.random.randn(2,3)

np.random.randint(0, 10, (3,4))

np.random.seed(42)

**🔄 Broadcasting**

a + 5 # Scalar broadcast

a + b # Compatible shapes

**🧱 Linear Algebra**

np.linalg.inv(a)

np.linalg.det(a)

np.linalg.eig(a)

np.linalg.solve(a, b)

**💾 Input/Output**

np.loadtxt("file.txt")

np.savetxt("out.txt", a)

np.save("array.npy", a)

np.load("array.npy")

**✅ Tips**

* Use .copy() when needed to avoid modifying original arrays.
* Prefer vectorized operations over loops for speed.

Would you like this cheat sheet in **PDF**, **Word**, or **HTML** format for download or printing?