```
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

4. Copying, Sorting & Reshaping

Description

NumPy provides various tools to **copy**, **sort**, and **reshape** arrays for data manipulation and analysis.
These operations are essential for **data organization**, **dimensional transformations**, and **memory control**.

Common Functions

Function	Description	Example Usage
`copy()`	Creates a deep copy of an array (independent of original)	`b = a.copy()`
`view()`	Creates a shallow copy (shares data with original)	`b = a.view()`
`sort()`	Sorts array elements along a specified axis	`np.sort(a)`
`flatten()`	Converts multi-dimensional array into 1D	`a.flatten()`
`T`	Transposes array (rows ↔ columns)	`a.T`
`reshape()`	Changes the shape without changing data	`a.reshape(2, 3)`
`resize()`	Changes shape and size (modifies original array)	`a.resize(3, 2)`

I. np.copy(arr)

Creates a deep copy — new memory space.

In [5]:
```
a = np.array([1,2,3])
b = np.copy(a)
b[0] = 99
print(a)
print(b) ### its affecting only dupicated data not effecyed to origiinal data
```

```
[1 2 3]
[99  2  3]
```

II. arr.view()

Creates a shallow copy (changes reflect on both).

In [8]:
```
v = a.view()
v[1] = 10
```

```
print(v)
print(a)  # Affected [1 10 3] in both original and duplicate data.
```

```
[ 1 10  3]
[ 1 10  3]
```

III. arr.sort(axis=0 or 1)

Sorts array in-place along an axis.

In [17]:
```
m = np.array([[3,1,2],[9,7,8]])
m.sort(axis=1)
print(m)
```

```
[[1 2 3]
 [7 8 9]]
```

IV. arr.flatten()

Returns a 1D copy of any array (no dimension nesting).

In [19]:
```
flat = m.flatten()
print(flat)
```

```
[1 2 3 7 8 9]
```

V. arr.reshape(rows, cols)

Changes the dimension but keeps total element count constant.

In [20]:
```
reshaped = np.arange(12).reshape(3,4)
print(reshaped)
```

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

VI. arr.T

Returns transpose — flips rows and columns.

In [22]:
```
print(reshaped.T)
```

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

VII. arr.resize(new_shape)

Modifies the original array's shape in-place (fills with zeros if needed).

In [24]:
```
reshaped.resize((2,6))
print(reshaped)
```

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

5. Adding & Removing Elements

Description

NumPy provides functions to **dynamically modify arrays** by adding, inserting, or deleting elements.
These operations create **new arrays** since NumPy arrays have **fixed size** once created.

Common Functions

Function	Description	Example Usage
`np.append()`	Adds elements to the end of an array	`np.append(a, [7, 8])`
`np.insert()`	Inserts values at a specific position	`np.insert(a, 1, 99)`
`np.delete()`	Deletes elements by index	`np.delete(a, [0, 2])`

I. np.append(arr, values, axis=None)

Adds new values at the end (creates new array).

append() always flattens arrays unless axis is specified.

In [25]:
```
arr = np.array([1,2,3])
arr2 = np.append(arr, [4,5])
print(arr2)
```

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

II. np.insert(arr, index, values, axis=None)

Inserts elements before the specified index.

In [26]:
```
arr = np.array([10,20,30])
inserted = np.insert(arr, 1, 15)
print(inserted)
```

```
[10 15 20 30]
```

III. np.delete(arr, index, axis=None)

Deletes elements along a given axis.

In numpy.delete(), for a two-dimensional array:

If axis=0, you delete rows. The index specified in the obj parameter refers to which row(s) to remove.

If axis=1, you delete columns. The index specified in the obj parameter refers to which column(s) to remove.

In [32]:
```
arr2d = np.array([[1,2,3],[4,5,6]])
deleted = np.delete(arr2d, 1, axis=0)
delete=np.delete(arr2d,1,axis=1)
print("deleted in columns using axis 1:")
print(delete)
print("deleted in columns using axis 1:",deleted)
```

```
deleted in columns using axis 1:
[[1 3]
 [4 6]]
deleted in columns using axis 1: [[1 2 3]]
```

6. Combining & Splitting Arrays

Description

NumPy allows you to **combine multiple arrays** into one or **split a large array** into smaller subsets.
These operations are essential for **data preprocessing**, **batch processing**, or **merging datasets**.

Common Functions

Function	Description	Example Usage
`np.concatenate()`	Joins two or more arrays along an axis	`np.concatenate((a, b), axis=0)`
`np.vstack()`	Stacks arrays vertically (row-wise)	`np.vstack((a, b))`
`np.hstack()`	Stacks arrays horizontally (column-wise)	`np.hstack((a, b))`
`np.split()`	Splits array into multiple sub-arrays	`np.split(a, 2)`

Function	Description	Example Usage
`np.hsplit()`	Splits array horizontally (by columns)	`np.hsplit(a, 2)`
`np.vsplit()`	Splits array vertically (by rows)	`np.vsplit(a, 2)`

I. np.concatenate((arr1, arr2), axis=0 or 1)

Merges arrays along rows (axis=0) or columns (axis=1).

In [35]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6]])
combined = np.concatenate((a,b), axis=0)
print(combined)
```

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

II. np.split(arr, num_splits)

Splits an array into multiple subarrays of equal size.

In [39]:
```
data = np.arange(10)
parts = np.split(data, 5)
print(parts)
```

```
[array([0, 1]), array([2, 3]), array([4, 5]), array([6, 7]), array([8, 9])]
[0 1 2 3 4 5 6 7 8 9]
```

III. np.hsplit(arr, num)

Splits horizontally (by columns).

In [80]:
```
arr1=np.arange(16).reshape(2,8)
left,right=np.hsplit(arr1,2)
print(left)
print(right)
```

```
[[ 0  1  2  3]
 [ 8  9 10 11]]
[[ 4  5  6  7]
 [12 13 14 15]]
```

In [81]:
```
# Create two 2x2 arrays
a = np.array([[1, 2],
              [3, 4]])

b = np.array([[5, 6],
              [7, 8]])
```

```python
# 1 Combine arrays vertically (along rows)
vertical = np.concatenate((a, b), axis=0)
print("Vertical Stack:\n", vertical)

# 2 Combine arrays horizontally (along columns)
horizontal = np.concatenate((a, b), axis=1)
print("\nHorizontal Stack:\n", horizontal)

# 3 Split array into equal parts
split_arr = np.split(vertical, 2)
print("\nSplit Arrays:")
for part in split_arr:
    print(part)

# 4 Horizontal split
h_split = np.hsplit(horizontal, 2)
print("\nHorizontal Split:")
for part in h_split:
    print(part)
```

```
Vertical Stack:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]

Horizontal Stack:
 [[1 2 5 6]
 [3 4 7 8]]

Split Arrays:
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]

Horizontal Split:
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]
```

7. NummPy Indexing & Slicing

📘 Overview

Efficiently access, extract, and manipulate specific parts of arrays — critical for **data cleaning, feature engineering, and model preparation**.

🔷 Basic Indexing

Concept	Syntax	Description	Example	Output
Access 1D element	`'arr[i]'`	Selects i-th element	`'arr[2]'`	Single value
Access 2D element	`'arr[i, j]'`	Selects element at row *i*, column *j*	`'arr[1, 2]'`	Single value
Negative index	`'arr[-1]'`	Selects last element	`'arr[-1]'`	Last element

✅ **Use Case:** Accessing specific data points in a table-like dataset.

In [51]:
```
arr = np.array([
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
])

print("Original Array:\n", arr)
print("Element at [0,1]:", arr[0, 1])
print("Element at [2,2]:", arr[2, 2])
print("Last element:", arr[-1, -1])
```

```
Original Array:
 [[10 20 30]
 [40 50 60]
 [70 80 90]]
Element at [0,1]: 20
Element at [2,2]: 90
Last element: 90
```

I. Basic Slicing (Rows, Columns, Subarrays)

Operation	Syntax	Meaning	Example Output
Row range	`'arr[0:2, :]'`	Rows 0–1, all columns	`[[10 20 30], [40 50 60]]`
Column range	`'arr[:, 1:3]'`	All rows, columns 1–2	`[[20 30], [50 60], [80 90]]`
Subarray	`'arr[0:2, 0:2]'`	Rows 0–1, columns 0–1	`[[10 20], [40 50]]`
Step slicing	`'arr[::2, ::2]'`	Every 2nd row and 2nd column	`[[10 30], [70 90]]`

✅ **Use Case:** Extract feature subsets, first N rows, or specific columns.

In [52]:
```
print("First 2 rows:\n", arr[0:2, :])
print("Last 2 columns:\n", arr[:, 1:3])
```

```
print("Top-left 2x2 block:\n", arr[0:2, 0:2])
print("Every 2nd element:\n", arr[::2, ::2])
```

```
First 2 rows:
 [[10 20 30]
 [40 50 60]]
Last 2 columns:
 [[20 30]
 [50 60]
 [80 90]]
Top-left 2x2 block:
 [[10 20]
 [40 50]]
Every 2nd element:
 [[10 30]
 [70 90]]
```

II. Negative Indexing

Type	Syntax	Description	Example Output
Last row	`'arr[-1]'`	Selects last row	`[70 80 90]`
Last column	`'arr[:, -1]'`	Selects last column	`[30 60 90]`
Bottom-right block	`'arr[-2:, -2:]'`	Last 2 rows & columns	`[[50 60], [80 90]]`

✅ **Use Case:** Quick access to recent or trailing data (e.g., last week, last month).

In [61]:
```
print("original array:",arr)
print("Last row:\n", arr[-1])
print("Last column:\n", arr[:, -1])
print("Bottom-right 2x2 block:\n", arr[-2:, -2:])
```

```
original array: [[10 20 30]
 [40 50 60]
 [70 80 90]]
Last row:
 [70 80 90]
Last column:
 [30 60 90]
Bottom-right 2x2 block:
 [[50 60]
 [80 90]]
```

8. Advanced Indexing (Fancy Indexing)

Use lists or arrays of indices to select arbitrary elements.

Concept	Syntax	Description	Example Output
Select specific rows	`'arr[[0, 2]]'`	Rows 0 and 2	`[[10 20 30], [70 80 90]]`

Concept	Syntax	Description	Example Output
Select specific columns	`'arr[:, [1, 2]]'`	Columns 1 and 2	`[[20 30], [50 60], [80 90]]`
Select custom subarray	`'arr[np.ix_([0,2], [1,2])]'`	Rows 0 & 2, Columns 1 & 2	`[[20 30], [80 90]]`

✅ **Use Case:** Select non-contiguous features or records — e.g., columns ['Age', 'Salary'] from selected customers.

In [62]:
```
print("original array:",arr)
rows = [0, 2]
cols = [1, 2]
print("Rows 0 & 2, Cols 1 & 2:\n", arr[np.ix_(rows, cols)])
```

```
original array: [[10 20 30]
 [40 50 60]
 [70 80 90]]
Rows 0 & 2, Cols 1 & 2:
 [[20 30]
 [80 90]]
```

A. Conditional (Boolean) Indexing

Condition Type	Syntax	Meaning	Example Output
Simple condition	`'arr[arr > 50]'`	Elements > 50	`[60 70 80 90]`
Combined (AND)	`'arr[(arr >= 30) & (arr <= 80)]'`	30 ≤ x ≤ 80	`[30 40 50 60 70 80]`
Combined (OR)	`'arr[(arr < 20) | (arr > 80)]'`	x < 20 or x > 80	`[10 90]`
Negation	`'arr[~(arr > 50)]'`	NOT condition	`[10 20 30 40 50]`

✅ **Use Case:** Data filtering — extract elements, rows, or features based on logical conditions.

In [64]:
```
print("original array:",arr)
print("Elements > 50:\n", arr[arr > 50])
print("Elements between 30 and 80:\n", arr[(arr >= 30) & (arr <= 80)])
print("Elements < 20 or > 80:\n", arr[(arr < 20) | (arr > 80)])
print("Negation (NOT > 50):\n", arr[~(arr > 50)])
```

```
original array: [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
Elements > 50:
 [999 999 999 999  60  70  80  90]
Elements between 30 and 80:
 [30 60 70 80]
Elements < 20 or > 80:
 [999 999 999 999  90]
Negation (NOT > 50):
 [30]
```

B. Extracting Rows and Columns

Action	Syntax	Output Example
Select single row	`'arr[1, :]'`	`[40 50 60]`
Select single column	`'arr[:, 2]'`	`[30 60 90]`
Select multiple rows/columns	`'arr[np.ix_([0, 2], [1, 2])]'`	`[[20 30], [80 90]]`

✅ **Use Case:** Retrieve specific rows/columns when preprocessing datasets for ML models.

In [65]:
```
print("original array:",arr)
print("2nd row:\n", arr[1, :])
print("3rd column:\n", arr[:, 2])
print("Rows 0 & 2, Columns 1 & 2:\n", arr[np.ix_([0, 2], [1, 2])])
```

```
original array: [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
2nd row:
 [999 999  60]
3rd column:
 [30 60 90]
Rows 0 & 2, Columns 1 & 2:
 [[999  30]
 [ 80  90]]
```

C. Step and Reversed Slicing

Operation	Syntax	Description	Output
Every 2nd row	`'arr[::2, :]'`	Selects alternate rows	`[[10 20 30], [70 80 90]]`
Reverse rows	`'arr[::-1, :]'`	Reverses row order	`[[70 80 90], [40 50 60], [10 20 30]]`
Reverse columns	`'arr[:, ::-1]'`	Reverses column order	`[[30 20 10], [60 50 40], [90 80 70]]`

✅ **Use Case:** Time-series reversal, data reordering, sampling.

In [66]:
```
print("original array:",arr)
print("Every 2nd row:\n", arr[::2, :])
print("Reversed rows:\n", arr[::-1, :])
print("Reversed columns:\n", arr[:, ::-1])
```

```
original array: [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
Every 2nd row:
 [[999 999  30]
 [ 70  80  90]]
Reversed rows:
 [[ 70  80  90]
 [999 999  60]
 [999 999  30]]
Reversed columns:
 [[ 30 999 999]
 [ 60 999 999]
 [ 90  80  70]]
```

D. Views vs Copies (Important!)

Type	Behavior	Example	Effect on Original
Slice (view)	Shares memory	`'subset = arr[0:2, 0:2]'`	✅ Changes affect original
Fancy/boolean (copy)	Independent memory	`'subset = arr[arr > 50]'`	❌ Changes don't affect original
Explicit copy	Force new memory	`'subset = arr[0:2, 0:2].copy()'`	Safe copy

✅ **Use Case:** Use `.copy()` when extracting subsets for isolated analysis or model testing,

In [68]:
```
subset = arr[0:2, 0:2]
subset[:] = 999
print("Subset (View):\n", subset)
print("Original after view modification:\n", arr)

# Create a copy safely
copy_subset = arr[0:2, 0:2].copy()
copy_subset[:] = 555
print("Copied subset:\n", copy_subset)
print("Original unaffected:\n", arr)
```

```
Subset (View):
 [[999 999]
 [999 999]]
Original after view modification:
 [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
Copied subset:
 [[555 555]
 [555 555]]
Original unaffected:
 [[999 999  30]
 [999 999  60]
 [ 70  80  90]]
```

--> Data Science Use Cases

Scenario	Indexing Type	Example
Extract first 10 records	Row slicing	`'arr[:10, :]'`
Select last 5 rows	Negative slicing	`'arr[-5:, :]'`
Select features (columns)	Column slicing	`'arr[:, [1, 3, 5]]'`
Filter by value	Boolean indexing	`'arr[arr[:, 2] > 50000]'`
Remove outliers	Boolean mask	`'arr[arr < threshold]'`
Sampling every 5th row	Step slicing	`'arr[::5, :]'`

✅ **Use Case:** Core step in EDA, feature selection, and model training pipelines.

In [86]:
```
data = np.arange(1, 41).reshape(10, 4)
print("Dataset (10x3):\n", data)

# Extract first 5 records
print("\nFirst 5 records:\n", data[:5, :])

# Last 3 records
print("\nLast 3 records:\n", data[-3:, :])

# Select feature columns (Age & Salary)
print("\nSelect columns 0 & 2:\n", data[:, [0, 2]])

# Filter elements > 20
print("\nElements > 20:\n", data[data > 20])

# Step slicing (every 2nd record)
print("\nEvery 2nd record:\n", data[::2, :])
```

```
Dataset (10x3):
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]
 [25 26 27 28]
 [29 30 31 32]
 [33 34 35 36]
 [37 38 39 40]]

First 5 records:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]

Last 3 records:
 [[29 30 31 32]
 [33 34 35 36]
 [37 38 39 40]]

Select columns 0 & 2:
 [[ 1  3]
 [ 5  7]
 [ 9 11]
 [13 15]
 [17 19]
 [21 23]
 [25 27]
 [29 31]
 [33 35]
 [37 39]]

Elements > 20:
 [21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]

Every 2nd record:
 [[ 1  2  3  4]
 [ 9 10 11 12]
 [17 18 19 20]
 [25 26 27 28]
 [33 34 35 36]]
```

--> Axis Concept Reference

Axis	Meaning	Example Operation	Description
`'axis=0'`	Column-wise	`'np.sum(arr, axis=0)'`	Operates vertically (down each column)
`'axis=1'`	Row-wise	`'np.sum(arr, axis=1)'`	Operates horizontally (across each row)

🧩 **Remember:**

- `'axis=0'` → Down the rows (columns are fixed)
- `'axis=1'` → Across the columns (rows are fixed)

🔷 Quick Summary Table

Type	Purpose	Returns	Typical Use
Basic Indexing	Select element(s)	Scalar	Direct access
Slicing	Select ranges	View	Subarray extraction
Fancy Indexing	Arbitrary picks	Copy	Non-contiguous selection
Boolean Indexing	Conditional filtering	Copy	Data cleaning/filtering
Step Slicing	Skipped selection	View	Sampling
Negative Indexing	From end	View	Last rows/cols
Axis Control	Direction of operation	-	Row/column aggregation

🧠 Key Takeaways

- `'arr[start:end, :] → row selection'`
- `'arr[:, start:end] → column selection'`
- **Views** share memory → efficient but risky for accidental edits.
- **Copies** are independent → safe for analysis.
- Master `'axis'` and conditional indexing — it's the foundation for **Pandas**, **machine learning preprocessing**, and **vectorized data filtering**.

✅ **Conclusion:**
Efficient **indexing and slicing** unlock NumPy's real power — enabling you to manipulate massive datasets in milliseconds.
It's one of the top 5 must-master skills for every **data scientist** working with Python.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: