# 16. Data Type Control (NumPy)

## 1. Topic Overview

### What this topic is

Data Type Control means controlling **how numbers are stored in memory** inside NumPy arrays.

Each NumPy array has **one fixed data type (dtype)**.

### Why it exists

Computers store data in memory.
Different data types use different memory sizes.
Correct dtype gives:

- Less memory usage
- Faster computation
- Correct numerical results

### One real-world analogy

Think of containers:

- Small bottle for water
- Big tank for fuel
  Using the wrong container wastes space or overflows.

## 2. Core Theory (Deep but Clear)

NumPy arrays are:

- Homogeneous (one dtype only)
- Stored in contiguous memory

- Optimized for math operations

## Internal NumPy view

Each array has:

- **shape** → how many rows and columns
- **dtype** → how each element is stored
- **itemsize** → bytes per element
- **memory block** → continuous memory

If dtype is wrong:

- Memory waste happens
- Precision loss happens
- ML models behave badly

## Sub-topics in Data Type Control

1. `dtype`
2. Common numeric dtypes
3. `astype()`
4. `itemsize`
5. Type inference
6. Overflow and precision issues

# 3. Syntax & Examples

## 3.1 `dtype`

### What it is

Defines the type of each element in the array.

## Basic syntax

```
import numpy as np

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

## Example 1

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

### Output

```
int64
```

### Explanation

- NumPy inferred type automatically
- Uses platform default integer

## Example 2

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

### Output

```
[1. 2. 3.]
float32
```

### Explanation

- Integers converted to floats
- Each value stored in 4 bytes

## 3.2 Common Numeric dtypes

| Type | Meaning | Bytes |
| --- | --- | --- |
| int8 | small integers | 1 |
| int32 | standard integers | 4 |
| int64 | large integers | 8 |
| float32 | decimal numbers | 4 |
| float64 | high precision | 8 |

### Example

```
arr = np.array([1, 2, 3], dtype=np.int8)
print(arr)
print(arr.itemsize)
```

#### Output

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

#### Explanation

- Each element uses 1 byte
- Very memory efficient
- Risk of overflow

## 3.3 `astype()`

### What it does

Changes dtype of an existing array.
Creates a **new copy**.

## Syntax

```
new_arr = arr.astype(np.float32)
```

## Example 1

```
arr = np.array([1, 2, 3])
new_arr = arr.astype(np.float32)

print(arr.dtype)
print(new_arr.dtype)
```

### Output

```
int64
float32
```

### Explanation

- Original array unchanged
- New array created

## Example 2

```
arr = np.array([1.7, 2.3, 3.9])
new_arr = arr.astype(int)

print(new_arr)
```

### Output

```
[1 2 3]
```

### Explanation

- Decimal part removed
- No rounding
- Dangerous in ML pipelines

## 3.4 `itemsize`

### What it is

Memory used by **one element**.

### Example

```
arr = np.array([1, 2, 3], dtype=np.float64)
print(arr.itemsize)
```

#### Output

```
8
```

#### Explanation

- Each value uses 8 bytes
- Large arrays become heavy

## 3.5 Type Inference

NumPy picks dtype based on input.

### Example

```
arr = np.array([1, 2.5, 3])
print(arr.dtype)
```

#### Output

```
float64
```

#### Explanation

- Mixed types
- NumPy chooses safest type

## 3.6 Overflow and Precision

### Example (Overflow)

```
arr = np.array([120, 121], dtype=np.int8)
print(arr + 10)
```

#### Output

```
[-126 -125]
```

#### Explanation

- int8 range exceeded
- Values wrapped
- Silent error

# 4. Why This Matters in Data Science

## Data cleaning

- Wrong dtype breaks missing value handling
- Strings stored as numbers cause errors

## Feature engineering

- Categorical encoding needs correct dtype
- Scaling fails if integers are used

## Model input preparation

- Deep learning models expect float32
- Wrong dtype slows GPU training

## ML / DL pipelines

- Overflow gives wrong gradients
- Precision loss ruins model accuracy

## If you don’t understand this

- Models train but give wrong results
- Memory usage explodes
- Bugs become invisible

# 5. Common Mistakes (VERY IMPORTANT)

1. Using default dtype blindly
   Cause: Trusting NumPy too much
   Fix: Always check `dtype`
2. Converting floats to int unknowingly
   Cause: `astype(int)`
   Fix: Convert only after validation
3. Ignoring overflow
   Cause: Small integer types
   Fix: Use `int32` or `int64`
4. Mixing numeric and string data
   Cause: Dirty datasets
   Fix: Clean before array creation
5. Using float64 everywhere
   Cause: Habit
   Fix: Use `float32` for ML

# 6. Performance & Best Practices

## When this is fast

- Correct dtype
- Contiguous memory
- Vectorized operations

## When this is slow

- Unnecessary type conversions
- Large float64 arrays
- Repeated `astype()` calls

## Memory warnings

- float64 doubles memory vs float32
- int8 can overflow silently

# 7. Practice Problems

## Easy (5)

1. Create an array of 100 zeros using `float32`
2. Check dtype and itemsize of an integer array
3. Convert a float array to int
4. Identify dtype chosen by NumPy for mixed input
5. Create an array with `int16` and print memory usage

## Medium (7)

1. Convert dataset values to `float32` for ML input
2. Detect overflow in small integer arrays
3. Reduce memory of a large array safely
4. Compare memory of float32 vs float64

5. Fix dtype issues in a noisy dataset
6. Prepare model input array with correct dtype
7. Convert categorical labels to integers

## Hard (5)

1. Debug wrong predictions caused by dtype
2. Fix precision loss in probability outputs
3. Optimize memory for million-row dataset
4. Detect silent overflow in pipeline
5. Prepare NumPy arrays for GPU training

## Industry-Level Tasks (3)

1. Convert raw CSV data into ML-ready NumPy arrays
   Input: mixed types
   Output: optimized dtype arrays
2. Memory optimization for real-time inference system
   Constraint: limited RAM
3. Fix training instability caused by dtype mismatch
   Scenario: deep learning model

# 8. Mini Checklist

- NumPy arrays have one dtype only
- dtype controls memory and speed
- Always check `dtype` and `itemsize`
- `astype()` creates a copy
- float32 is standard for ML
- Small integers can overflow
- Wrong dtype breaks models silently

# 16 (Part 2). Advanced Data Type Control (Mandatory for Data Science)

## 1. Topic Overview

### What this topic is

These are **advanced NumPy dtypes** used for:

- dates
- time differences
- boolean masks
- mixed structured data
- raw Python objects

### Why it exists

Real datasets are not only numbers.
They contain:

- dates
- categories
- flags
- mixed columns

NumPy needs special dtypes to handle them safely.

### One real-world analogy

Excel columns.
Date column is not same as number column.
Each needs different rules.

# 2. Core Theory (Deep but Clear)

NumPy dtype system is larger than `int` and `float`.

Important advanced dtypes:

1. `bool`
2. `datetime64`
3. `timedelta64`
4. `object`
5. Structured dtypes
6. `view()` vs `astype()`

If you ignore these:

- Time-series analysis breaks
- Masking becomes slow
- Pipelines crash silently

# 3. Syntax & Examples

## 3.1 Boolean dtype ( `bool` )

### What it is

Stores `True` or `False`.
Uses **1 byte per value**.

Used heavily for filtering.

### Syntax

```
np.array(data, dtype=bool)
```

## Example 1

```
import numpy as np

arr = np.array([1, 0, 5, 0], dtype=bool)
print(arr)
print(arr.dtype)
```

### Output

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

### Explanation

- Non-zero becomes `True`
- Zero becomes `False`

## Example 2 (Masking)

```
data = np.array([10, 20, 30, 40])
mask = data > 25

print(mask)
print(data[mask])
```

### Output

```
[False False  True  True]
[30 40]
```

### Explanation

- Boolean array used as filter
- Core Data Science operation

## 3.2 `datetime64`

### What it is

Stores dates and timestamps.
Much faster than Python `datetime`.

### Syntax

```
np.array(dates, dtype='datetime64')
```

### Example 1

```
dates = np.array(['2024-01-01', '2024-01-10'], dtype='datetime64')
print(dates)
print(dates.dtype)
```

#### Output

```
['2024-01-01' '2024-01-10']
datetime64[D]
```

#### Explanation

- Stored as integers internally
- Unit is days

### Example 2

```
print(dates[1] - dates[0])
```

#### Output

```
9 days
```

#### Explanation

- Vectorized date math
- No loops needed

## 3.3 `timedelta64`

### What it is

Stores time difference.
Used with `datetime64` .

### Example

```
delta = np.array([1, 3, 7], dtype='timedelta64[D]')
print(delta)
```

#### Output

```
[1 3 7]
```

#### Explanation

- Represents day differences
- Used in time-based features

## 3.4 `object` dtype

### What it is

Stores **raw Python objects**.
Very slow.
Avoid unless required.

## Example

```
arr = np.array(['apple', 10, 3.5])
print(arr.dtype)
```

### Output

```
object
```

### Explanation

- Mixed types force object dtype
- No vectorization
- Memory heavy

# 3.5 Structured dtypes

## What it is

Multiple fields per element.
Like rows with fixed schema.

## Example

```
dtype = [('id', 'int32'), ('score', 'float32')]
data = np.array([(1, 90.5), (2, 85.0)], dtype=dtype)

print(data)
print(data['score'])
```

### Output

```
[(1, 90.5) (2, 85. )]
[90.5 85. ]
```

### Explanation

- Each element has structure

- Used in low-level pipelines

## 3.6 `view()` vs `astype()` (CRITICAL)

### `astype()`

- Changes dtype
- Creates copy

### `view()`

- Reinterprets memory
- No copy

## Example

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

view_arr = arr.view(np.float32)
print(view_arr)
```

### Output

```
[1.4012985e-45 2.8025969e-45 4.2038954e-45]
```

### Explanation

- Same memory
- Interpreted differently
- Dangerous if misunderstood

# 4. Why This Matters in Data Science

## Data cleaning

- Boolean masks remove bad rows
- Object dtype signals dirty data

## Feature engineering

- Date differences create time features
- Flags stored as bool

## Model input preparation

- Object dtype breaks ML models
- Dates must be numeric or encoded

## ML / DL pipelines

- Time-based splits rely on datetime
- Wrong dtype causes silent failure

## If you skip this

- Time series models fail
- Filtering becomes slow
- Pipelines become unstable

# 5. Common Mistakes (VERY IMPORTANT)

1. Leaving strings as `object`
   Cause: Direct CSV load
   Fix: Convert early
2. Using Python datetime instead of `datetime64`
   Cause: Habit
   Fix: Use NumPy datetime

3. Using `view()` without understanding memory
   Cause: Curiosity
   Fix: Prefer `astype()`
4. Boolean masks with wrong shape
   Cause: Shape mismatch
   Fix: Always check shape
5. Structured dtypes for ML inputs
   Cause: Overengineering
   Fix: Flatten before modeling

# 6. Performance & Best Practices

## Fast

- bool masks
- datetime64 math
- numeric dtypes

## Slow

- object dtype
- Python loops
- mixed-type arrays

## Warnings

- object dtype disables vectorization
- view() can corrupt logic
- datetime unit mismatch causes bugs

# 7. Practice Problems

## Easy (5)

1. Create a boolean mask for values > mean

2. Convert string dates to `datetime64`
3. Subtract two date arrays
4. Detect object dtype in an array
5. Create a boolean array from integers

## Medium (7)

1. Filter dataset using boolean masks
2. Create time-gap features from dates
3. Convert object arrays to numeric safely
4. Identify slow operations caused by dtype
5. Fix shape mismatch in boolean indexing
6. Encode date differences as integers
7. Remove object dtype from ML inputs

## Hard (5)

1. Debug model crash caused by object dtype
2. Optimize time-series feature generation
3. Replace Python datetime with NumPy datetime
4. Detect memory misuse from `view()`
5. Clean mixed-type NumPy arrays

## Industry-Level Tasks (3)

1. Build time-based features for churn prediction
   Input: user signup dates
   Output: numeric features
2. Optimize filtering on 10M rows using boolean masks
3. Fix production pipeline slowed by object dtype arrays

# 8. Mini Checklist

- Boolean masks are core to filtering
- datetime64 is mandatory for time data
- object dtype is dangerous
- Structured dtypes are not for ML models
- `astype()` copies, `view()` does not
- Always remove object dtype before modeling