

# Python Data Types: Comprehensive Notes

## Introduction

Data types in Python define the type of data a variable can hold. They are critical in determining the operations that can be performed on the data and the structure of the program. Python provides a rich set of built-in data types, offering flexibility, ease of use, and functionality for various programming tasks.

---

## 1. Numeric Data Types

Numeric data types represent numerical values. Python provides three main numeric types:

### 1.1. Integer (int)

#### Definition:

Represents whole numbers without decimal points. They can be of unlimited length in Python (subject to memory constraints).

#### Description:

- Examples: `-2`, `0`, `5`, `100`
- Stored as binary values internally.

#### Use Cases:

- Counting items.
- Indexing arrays or lists.
- Representing IDs or keys.

#### Applications:

- Mathematical calculations.

- Iterations in loops.

**Advantages:**

- Simple and efficient.
- Unlimited precision.

**Disadvantages:**

- Cannot represent fractional values.

**Time Complexity:**

- Basic operations like addition, subtraction, multiplication:  **$O(1)$** .
- Division:  **$O(\log(n))$** , due to potential large integers.

**Inbuilt Methods:**

- `bit_length()`: Returns the number of bits required to represent the integer.
- `to_bytes()`: Converts an integer to a byte array.
- `from_bytes()`: Converts a byte array back to an integer.

**Implementation:**

```
x = 10
print(x.bit_length()) # Output: 4
```

---

## 1.2. Float (float)

**Definition:**

Represents real numbers (with decimal points) and is implemented as double-precision floating-point numbers.

**Description:**

- Examples: `3.14`, `-0.001`, `2.0`
- Stored using IEEE 754 standard.

**Use Cases:**

- Representing measurements.

- Scientific computations.

**Applications:**

- Data analysis.
- Graphics (coordinates).
- Physics simulations.

**Advantages:**

- Can represent a wide range of values.

**Disadvantages:**

- Subject to rounding errors.
- Limited precision (~15-16 decimal places).

**Time Complexity:**

- Basic operations:  **$O(1)$** .

**Inbuilt Methods:**

- `is_integer()`: Checks if the float is equivalent to an integer.

**Implementation:**

```
y = 3.14159
print(y.is_integer()) # Output: False
```

---

## 1.3. Complex (complex)

**Definition:**

Represents complex numbers with real and imaginary parts.

**Description:**

- Examples: `3+4j`, `-2-7j`
- Both parts are floats.

**Use Cases:**

- Electrical engineering.
- Signal processing.

**Applications:**

- Simulating dynamic systems.

**Advantages:**

- Direct support for complex numbers.

**Disadvantages:**

- Niche use cases.

**Inbuilt Methods:**

- `real`: Returns the real part.
- `imag`: Returns the imaginary part.

**Implementation:**

```
z = 3 + 4j
print(z.real)    # Output: 3.0
print(z.imag)    # Output: 4.0
```

---

## 2. Sequence Data Types

Sequence types represent ordered collections of items.

### 2.1. String (str)

**Definition:**

Represents a sequence of characters.

**Description:**

- Immutable.

- Can contain any Unicode characters.

**Use Cases:**

- Storing textual data.
- Representing file paths or URLs.

**Applications:**

- Web development.
- Text processing.

**Advantages:**

- Rich set of methods.

**Disadvantages:**

- Memory-intensive for large texts.

**Time Complexity:**

- Access:  $O(1)$
- Concatenation:  $O(n)$

**Inbuilt Methods:**

- `lower()`, `upper()`, `split()`, `join()`, `strip()`

**Implementation:**

```
text = "Hello, World!"  
print(text.lower()) # Output: hello, world!
```

---

## 2.2. List (list)

**Definition:**

A mutable sequence of items, which can be of different types.

**Description:**

- Examples: `[1, 2, 3]`, `['a', 'b']`

- Allows indexing and slicing.

#### Use Cases:

- Dynamic arrays.
- Collections of related items.

#### Applications:

- Data storage.
- Iterative processing.

#### Advantages:

- Flexible and dynamic.

#### Disadvantages:

- Memory overhead.

#### Time Complexity:

- Access:  $O(1)$
- Insertion/Deletion:  $O(n)$  (worst case)

#### Inbuilt Methods:

- `append()`, `pop()`, `sort()`, `reverse()`

#### Implementation:

```
items = [1, 2, 3]
items.append(4)
print(items) # Output: [1, 2, 3, 4]
```

---

## 2.3. Tuple (tuple)

#### Definition:

An immutable sequence of items.

**Description:**

- Examples: (1, 2, 3), ('a', 'b')

**Use Cases:**

- Fixed collections.
- Keys in dictionaries.

**Applications:**

- Ensuring data integrity.

**Advantages:**

- Immutable (safer).

**Disadvantages:**

- Cannot modify elements.

**Time Complexity:**

- Access:  $O(1)$

**Inbuilt Methods:**

- `count()`, `index()`

**Implementation:**

```
tup = (1, 2, 3)
print(tup.index(2)) # Output: 1
```

---

## 2.4. Range (range)

### Definition:

Represents an immutable sequence of numbers.

### Description:

- Examples: `range(0, 10)`

### Use Cases:

- Iterations in loops.

### Applications:

- Generating sequences.

### Advantages:

- Memory-efficient.

### Disadvantages:

- Read-only.

### Time Complexity:

- Access:  $O(1)$

### Implementation:

```
r = range(5)
print(list(r))  # Output: [0, 1, 2, 3, 4]
```

---



## 3. Mapping Data Types

### 3.1. Dictionary (dict)

#### Definition:

Represents key-value pairs.

#### Description:

- Examples: `{'a': 1, 'b': 2}`

#### Use Cases:

- Lookup tables.
- Storing configurations.

#### Applications:

- Data indexing.

#### Advantages:

- Fast lookups.

#### Disadvantages:

- Unordered (in versions <3.7).

#### Time Complexity:

- Access:  **$O(1)$**
- Insertion:  **$O(1)$**  (amortized)


#### Inbuilt Methods:

- `.get(key[, default])`: Returns the value for a key, or a default if the key is not found.
- `.keys()`, `.values()`, `.items()`: Return views of the dictionary's keys, values, or key-value pairs.

- `.update(other_dict)`: Updates the dictionary with key-value pairs from another dictionary.
- `.pop(key[, default])`: Removes the key and returns its value.

### Implementation:

python

 Copy code

```
# Creating a dictionary
person = {"name": "Aditya", "age": 20, "location": "India"}

# Accessing elements
print(person["name"]) # Output: Aditya

# Adding a new key-value pair
person["hobby"] = "Programming"

# Removing a key-value pair
del person["age"]

print(person) # Output: {'name': 'Aditya', 'location': 'India', 'hobby': 'Programming'}
```

---

## 4. Set Data Types

### 4.1. Set (set)

#### Definition:

Represents an unordered collection of unique elements.

**Description:**

- Examples: {1, 2, 3}

**Use Cases:**

- Removing duplicates.

**Applications:**

- Membership testing.

**Advantages:**

- Fast membership testing.

**Disadvantages:**

- Unordered.

**Time Complexity:**

- Access:  $O(1)$

**Inbuilt Methods:**

- `add()`, `remove()`, `union()`, `intersection()`

**Implementation:**

```
s = {1, 2, 3}
s.add(4)
print(s) # Output: {1, 2, 3, 4}
```

## 4.2. Frozen Set (`frozenset`)

**Definition:**

A `frozenset` is an immutable version of a set, meaning its elements cannot be added or removed after the frozenset is created.

Examples:

```
python

frozenset_example = frozenset([1, 2, 3, 4])
print(frozenset_example) # Output: frozenset({1, 2, 3, 4})
```

- Unlike `set`, a `frozenset` is hashable, making it eligible to be used as a key in a dictionary or as an element in another set.
- Maintains unique elements and does not allow duplicates.

**Use Cases:**

- Representing immutable collections of unique elements.
- Using as dictionary keys or elements in other sets.
- Working with data that must not change during program execution.

**Applications:**

- Ensuring data integrity in scenarios where a collection of items must remain constant.
- Building complex data structures where sets need to be nested.

**Advantages:**

- Hashable and immutable, suitable for use in hashed collections like dictionaries.
- Supports all set operations like union, intersection, and difference.

**Disadvantages:**

- Cannot modify elements, which limits its flexibility compared to `set`.

**Time Complexity:**

- Access:  **$O(1)$**  (average case for operations like membership testing).
- Union, Intersection, Difference:  **$O(\text{len}(s) + \text{len}(t))$** , where `s` and `t` are the sets involved.

**Inbuilt Methods:**

- `.union(other)`: Returns a new frozenset containing elements from both sets.
- `.intersection(other)`: Returns a frozenset of common elements.

- `.difference(other)`: Returns a frozenset with elements unique to the first set.
- `.isdisjoint(other)`: Checks if two sets have no common elements.

### Implementation:

```
python

a = frozenset([1, 2, 3])
b = frozenset([3, 4, 5])

# Union
print(a.union(b)) # Output: frozenset({1, 2, 3, 4, 5})

# Intersection
print(a.intersection(b)) # Output: frozenset({3})

# Difference
print(a.difference(b)) # Output: frozenset({1, 2})
```

---

## 5. Boolean Data Type

### 5.1. Boolean (`bool`)

#### Definition:

A `bool` represents one of two values: `True` or `False`. It is a subclass of `int`, where `True` is equivalent to `1` and `False` is equivalent to `0`.

#### Description:

Examples:

```
python

is_active = True
is_deleted = False
```

- Used for logical operations and control flow in programming.

**Use Cases:**

- Representing binary states, such as "on/off", "yes/no".
- Controlling loops and conditionals.

**Applications:**

- Decision-making in programs.
- Flags for conditional execution.

**Advantages:**

- Simplifies logical operations.
- Easy to use and integrate with other data types.

**Disadvantages:**

- Limited to binary states.

**Time Complexity:**

- Logical operations: **O(1)**.

**Inbuilt Methods:**

- `bool(x)`: Converts a value to a boolean (`True` or `False`).

### Implementation:

```
python

# Boolean evaluation
print(bool(0))    # Output: False
print(bool(1))    # Output: True
print(bool([]))   # Output: False
print(bool([1]))  # Output: True
```

---

## 6. Binary Data Types

### 6.1. Bytes (**bytes**)

#### Definition:

Immutable sequences of bytes, typically used to store binary data like images, files, and network communications.

#### Description:

Examples:

```
python

byte_seq = b"hello"
print(byte_seq)  # Output: b'hello'
```

- Each element in a **bytes** object is an integer between 0 and 255.

#### Use Cases:

- Storing and manipulating binary data.
- Encoding and decoding strings for network transmissions.

**Applications:**

- File I/O operations.
- Data serialization and deserialization.

**Advantages:**

- Immutable and lightweight.
- Efficient for binary data handling.

**Disadvantages:**

- Cannot modify individual bytes without creating a new object.

**Inbuilt Methods:**

- `.decode(encoding)`: Converts bytes to a string.
- `.hex()`: Returns the hexadecimal representation.

**Implementation:**

```
python

byte_seq = b"hello"
print(byte_seq.decode("utf-8")) # Output: hello
```

---



## 6.2. Bytearray (**bytearray**)

### Definition:

A mutable sequence of bytes.

### Description:

Examples:

```
python

mutable_byte_seq = bytearray(b"hello")
mutable_byte_seq[0] = 72 # ASCII for 'H'
print(mutable_byte_seq) # Output: bytearray(b'Hello')
```

### Use Cases:

- Modifying binary data.
- Efficient for mutable byte sequences.

### Applications:

- Real-time data manipulation.
- Streamlined I/O operations.

### Advantages:

- Mutable, allowing in-place changes.

### Disadvantages:

- Slightly more memory-intensive than **bytes**.

### Inbuilt Methods:

- **.append(x)**: Adds a byte.
- **.extend(iterable)**: Extends the sequence.

## Implementation:

```
python

ba = bytearray(b"abc")
ba.append(100) # ASCII for 'd'
print(ba)      # Output: bytearray(b'abcd')
```

---

## 6.3. Memoryview (**memoryview**)

### Definition:

Provides a view object that references another object's memory without copying it. Primarily used for performance optimization.

### Description:

Examples:

```
python

data = bytearray(b"hello")
mv = memoryview(data)
print(mv[0]) # Output: 104 (ASCII for 'h')
```

### Use Cases:

- Zero-copy data access.
- Efficient slicing and processing of large data.

### Applications:

- Data buffering in I/O operations.
- Shared memory systems.

### Advantages:

- Avoids redundant memory copies.
- Optimized for performance.

### Disadvantages:

- Requires familiarity with low-level memory operations.

### Implementation:

```
python

data = bytearray(b"hello")
mv = memoryview(data)
mv[0] = 72 # ASCII for 'H'
print(data) # Output: bytearray(b'Hello')
```

## 8. None Data Type

### 8.1. None (NoneType)

#### Definition:

**None** represents the absence of a value or a null value in Python. It is a singleton object of type **NoneType**.

#### Description:

Examples:

```
python

x = None
print(x is None) # Output: True
```

- Commonly used as a placeholder for optional arguments, uninitialized variables, or to indicate the end of a function that doesn't return a value explicitly.

#### Use Cases:

- Representing "nothingness" or "no value."

- Default return value for functions without `return` statements.
- Sentinel values in data structures.

### Applications:

- Simplifies conditional checks for missing or uninitialized data.
- Signals the end of data processing.

### Advantages:

- Clear and concise representation of null values.
- Enhances readability and intent of code.

### Disadvantages:

- Limited to signaling absence; not suitable for representing complex data states.

### Time Complexity:

- Comparisons and assignments: **O(1)**.

### Implementation:

```
python

# Function with no return value
def greet(name):
    print(f"Hello, {name}")

result = greet("Aditya")
print(result) # Output: None

# Placeholder
data = None
if data is None:
    print("Data is missing.")
```

---

## 9. Specialized Data Types

Python's standard library provides specialized data types in the `collections` module that extend the functionality of built-in types.

### 9.1. Named Tuple (`collections.namedtuple`)

#### Definition:

A `namedtuple` is a tuple with named fields, allowing access to elements using attribute names in addition to index positions.

#### Description:

Examples:

```
python

from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])
p = Point(10, 20)
print(p.x, p.y) # Output: 10 20
```

- Immutable like regular tuples.
- Enhances readability and intent in code by using meaningful names for elements.

#### Use Cases:

- Storing structured data in a lightweight way.
- Replacing simple classes in some scenarios.

#### Applications:

- Geometric points, database records, configuration settings.

#### Advantages:

- Lightweight and readable.
- Combines tuple immutability with named field access.

### Disadvantages:

- Limited to fixed field names and immutable fields.

### Implementation:

```
python

from collections import namedtuple

Student = namedtuple("Student", ["name", "age", "grade"])
s = Student(name="Aditya", age=20, grade="A")
print(s.name, s.age, s.grade) # Output: Aditya 20 A
```

---

## 9.2. Deque (**collections.deque**)

### Definition:

A **deque** (double-ended queue) is a thread-safe, generalization of stacks and queues that allows fast appends and pops from both ends.

### Description:

Examples:

```
python

from collections import deque

dq = deque([1, 2, 3])
dq.append(4)           # Add to the right
dq.appendleft(0)       # Add to the left
print(dq)              # Output: deque([0, 1, 2, 3, 4])
```

- Optimized for adding and removing elements from either end in **O(1)**.

**Use Cases:**

- Implementing stacks, queues, and other dynamic collections.

**Applications:**

- Real-time data processing.
- Sliding window computations.

**Advantages:**

- Efficient for insertions and deletions at both ends.
- Built-in support for thread-safe operations.

**Disadvantages:**

- Limited in use cases compared to other containers.

**Implementation:**

```
python

from collections import deque

dq = deque([1, 2, 3])
dq.pop()           # Removes from the right
dq.popleft()       # Removes from the left
print(dq)          # Output: deque([2])
```

## Homogeneous Data Types in Python

Homogeneous data types refer to data structures that contain elements of the same data type.

- **Arrays (from the NumPy library):** Arrays are specifically designed to store homogeneous data, primarily numerical data. They are efficient for numerical computations.

## Heterogeneous Data Types in Python

Heterogeneous data types refer to data structures that can contain elements of different data types.

- **Lists:** As mentioned earlier, lists can store a mix of data types, making them versatile for various applications.
- **Tuples:** Similar to lists, tuples can also store heterogeneous data. However, unlike lists, tuples are immutable, meaning their elements cannot be changed once the tuple is created.
- **Dictionaries:** Dictionaries store key-value pairs, where keys and values can be of different data types.

## Primitive and Non-Primitive Data Types in Python

### Primitive Data Types:

- **Numeric:**
  - int: Integer numbers (e.g., 10, -5, 0)
  - float: Floating-point numbers (e.g., 3.14, -2.5)
  - complex: Complex numbers (e.g., 2+3j)
- **Boolean:**
  - bool: True or False values
- **String:**
  - str: Sequence of characters (e.g., "Hello, world!")

### Non-Primitive Data Types:

- **List:** Ordered collection of items
- **Tuple:** Ordered, immutable collection of items



- **Set:** Unordered collection of unique items
- **Dictionary:** Unordered collection of key-value pairs

### Mutable Data Types:

- Lists
- Dictionaries
- Sets

### Immutable Data Types:

- Numbers (int, float, complex)
- Strings
- Tuples
- Booleans
- Frozen Sets

### Key Differences:

Feature	Mutable Data Types	Immutable Data Types
Value Changeability	Can be changed	Cannot be changed
Memory Allocation	New objects are created when modifications are made	New objects are created for any change
Efficiency	Less efficient for frequent modifications	More efficient, especially in multi-threaded environments
Use Cases	Dynamic data structures, where frequent changes are required	Static data structures, where data integrity is crucial