Introduction to Python's Core Data Types

January 8, 2025

Contents

1 Overview **2**

2 Key Terms and Concepts **3**

3 Data Type Summary **4**

4 Numeric Types **4**
- 4.1 `int` . . . 4
- 4.2 `float` . . . 5
- 4.3 `complex` . . . 5
- 4.4 `bool` . . . 5

5 Sequence Types **7**
- 5.1 `str` . . . 7
- 5.2 `list` . . . 7
- 5.3 `tuple` . . . 8
- 5.4 `bytes` and `bytearray` . . . 8

6 Set Types **10**
- 6.1 `set` . . . 10
- 6.2 `frozenset` . . . 10

7 Mapping Types **11**
- 7.1 `dict` . . . 11

8 `NoneType` (`None`) **11**

9 Type Conversion and Casting **12**

10 Mutability vs. Immutability **12**

11 Final Tips **12**

1 Overview

Python provides a rich set of built-in data types to handle a variety of use cases. They can be broadly categorized into:

- **Numeric Types**: `int`, `float`, `complex`, and `bool`.
- **Sequence Types**: `str`, `list`, `tuple`, `bytes`, `bytearray`.
- **Set Types**: `set`, `frozenset`.
- **Mapping Types**: `dict`.
- **Special Types**: `NoneType`, `memoryview`.

This document covers their key features, usage examples, and common pitfalls. For more advanced details, refer to Python's official documentation:

- https://docs.python.org/3/library/stdtypes.html
- https://docs.python.org/3/library/functions.html

2 Key Terms and Concepts

Immutable A data type is *immutable* if its value cannot be changed after it is created. Attempts to modify the object in-place will raise an error. In Python, examples include `int`, `float`, `complex`, `bool`, `str`, `tuple`, `frozenset`, `bytes`, and `NoneType`.

Mutable A data type is *mutable* if it can be changed in place. This means you can add, remove, or alter elements without creating a new object. Examples include `list`, `set`, `dict`, and `bytearray`.

Hashable An object is *hashable* if it has a hash value that never changes during its lifetime (so that it can be used as a key in a dictionary or stored in a set). Typically, all immutable built-in objects are hashable by default.

Insertion-ordered As of Python 3.7+, dictionaries (`dict`) maintain insertion order, meaning items are iterated in the order they were inserted.

Arbitrary precision Python's `int` type can grow to a very large size as needed (limited mostly by available memory), unlike many languages with fixed maximum integer sizes.

Floor division (`//`) Using `//` between two numbers divides them and rounds down (floors) to the nearest integer result.

Shallow copy vs. Deep copy A *shallow copy* of an object copies the outer container but references the same elements within. A *deep copy* recursively copies everything, resulting in completely independent nested objects.

3 Data Type Summary

Table 1: Data Type Summary

Data Type	Mutability	Notes
`int`	Immutable	Integral numbers (arbitrary precision)
`float`	Immutable	Floating-point numbers
`complex`	Immutable	Real + imaginary parts
`bool`	Immutable	True or False (`True`=1, `False`=0)
`str`	Immutable	Unicode text sequence
`list`	Mutable	Ordered sequence, can append/remove/sort
`tuple`	Immutable	Ordered sequence, cannot be changed
`set`	Mutable	Unique elements, no indexing
`frozenset`	Immutable	Like `set`, but cannot be changed
`dict`	Mutable	Key-value pairs, insertion-ordered
`bytes`	Immutable	Sequence of bytes (binary data)
`bytearray`	Mutable	Mutable sequence of bytes
`NoneType`	N/A	Absence of value (`None`)
`memoryview`	N/A	View of binary data without copying

4 Numeric Types

4.1 `int`

Whole numbers (positive, negative, or zero) without decimals. They have *arbitrary precision.*

```
x = 10
y = -5
```

Operations:

- Arithmetic: `+`, `-`, `*`, `//`, `%`, `**`.
- Comparison: `<`, `>`, `<=`, `>=`, `==`, `!=`.
- Conversion: `int("5")` converts a string to an integer (if valid).

Restrictions: Cannot mix non-numeric types directly:

```
5 + "hello" # Raises TypeError
```

4.2 `float`

Real numbers with decimals or scientific notation.

```
x = 10.5
y = 1.23e3 # 1.23 * 10^3
```

Operations: Same as `int`, plus math module functions:

```
import math
result = math.sqrt(16) # 4.0
```

Restrictions: Floating-point precision errors:

```
0.1 + 0.2 == 0.3 # False
```

4.3 `complex`

Numbers with a real and imaginary part.

```
z = 3 + 4j
```

Operations:

- Arithmetic: `+`, `-`, `*`, `/`.
- Attributes: `z.real`, `z.imag`.

Restrictions: Cannot compare complex numbers with `<` or `>`.

4.4 `bool`

`bool` is a subclass of `int`, where `True` = 1 and `False` = 0.

```
is_active = True
has_error = False
```

Usage: Commonly arises from comparisons or logical operations:

```
is_bigger = (5 > 3) # True


True + True # 2
False * 5 # 0
```

5 Sequence Types

5.1 `str`

Ordered, *immutable* sequences of Unicode characters.

```
s = "hello"
```

Operations:

- Concatenation: `"hello" + " world"` → `'hello world'`
- Repetition: `"ha" * 3` → `'hahaha'`
- Indexing: `s[0]` → `'h'`
- Negative Indexing: `s[-1]` → `'o'` (last char)
- Slicing: `s[1:4]` → `'ell'`
- Extended Slicing (step): `s[::2]` → `'hlo'`
- Methods: `s.upper()`, `s.lower()`, `s.replace()`, etc.

Restrictions: Strings are *immutable*:

```
s[0] = 'H' # Raises TypeError
```

5.2 `list`

Ordered, *mutable* sequences of elements.

```
l = [1, 2, 3]
```

Operations:

- Add/Remove: `l.append(4)`, `l.remove(2)`, `l.pop()`
- Sorting: `l.sort()`, `l.reverse()`
- Indexing and slicing: Similar to strings (including negative indices).

Restrictions: Mixing types may cause errors during certain operations:

```
sum(["1", 2]) # Raises TypeError
```

Shallow vs. Deep Copy

When copying lists, be mindful of nested structures:

```
import copy

original = [[1, 2], [3, 4]]
shallow_copy = copy.copy(original)
deep_copy = copy.deepcopy(original)

original[0][0] = 'X'

print(shallow_copy[0][0]) # 'X' (same inner list reference)
print(deep_copy[0][0]) # 1 (independent copy)
```

5.3 `tuple`

Ordered, *immutable* sequences.

```
t = (1, 2, 3)
```

Operations:

- Indexing and slicing: same as lists/strings.
- Count occurrences: `t.count(value)`
- Find index: `t.index(value)`

Restrictions: `tuple` is *immutable*:

```
t[0] = 5 # Raises TypeError
```

5.4 `bytes` and `bytearray`

`bytes`: An *immutable* sequence of bytes (often used for binary data).

```
b = b"hello" # type: bytes
```

bytearray: A *mutable* sequence of bytes.

```
ba = bytearray(b)
ba[0] = 72 # ASCII for 'H'
```

memoryview: Offers a way to access the memory of another object (e.g., bytes) without copying.

```
mview = memoryview(ba)
print(mview[0]) # 72
```

6 Set Types

6.1 `set`

Unordered, *mutable*, unique elements.

```
s = {1, 2, 3}
```

Operations:

- Add/Remove: `s.add(4)`, `s.remove(2)`
- Common operations:
  - `s.union(t)`, `s.intersection(t)`, `s.difference(t)`, `s.symmetric_difference(t)`
- Operators:
  - `s1  s2|` (union)
  - `s1 & s2` (intersection)
  - `s1 - s2` (difference)
  - `s1 ^ s2` (symmetric difference)

Restrictions: No indexing or slicing:

```
s[0] # Raises TypeError
```

6.2 `frozenset`

Immutable version of a set.

```
fs = frozenset([1, 2, 3])
```

Restrictions: Cannot modify elements after creation.

7 Mapping Types

7.1 `dict`

Key-value pairs, *mutable*, insertion-ordered (Python 3.7+).

```
d = {"name": "John", "age": 30}
```

Operations:

- Access: `d["name"]` or `d.get("name")`
- Add/Update: `d["city"] = "NY"`
- Methods:
  - `d.keys()`, `d.values()`, `d.items()`
  - `d.get(key, default)`
  - `d.setdefault(key, default)`
- Merge (Python 3.9+): `d1 | d2`

Restrictions: Keys must be *hashable* (no `list` or `dict` as keys).

8 `NoneType (None)`

`None` represents the absence of a value.

```
result = None
```

Usage:

- Often a default return value of functions with no explicit return.
- Always compare with `is`:

```
if result is None:
print("No value assigned yet.")
```

9 Type Conversion and Casting

Python provides several built-in functions for casting:

- `int()`, `float()`, `str()`, `bool()`
- Casting numeric types: `float(5)` → `5.0`, `int(5.9)` → `5`
- Converting containers: `list()`, `tuple()`, `set()`, etc.

```
num_str = "100"
num_int = int(num_str) # 100
num_float = float(num_int) # 100.0
```

10 Mutability vs. Immutability

- **Immutable**: `int`, `float`, `complex`, `bool`, `str`, `tuple`, `frozenset`, `bytes`, `NoneType`.
- **Mutable**: `list`, `set`, `dict`, `bytearray`.

(See Section **??** *Key Terms and Concepts* for definitions.)

11 Final Tips

- Pay attention to **immutability** vs. **mutability** when selecting data structures.
- Use **negative indices** and **step slicing** to your advantage with sequences.
- Use **shallow** vs. **deep copies** appropriately to avoid hidden side effects.
- Watch out for **floating-point precision** issues.
- Refer to the **official documentation** for comprehensive details:
  - https://docs.python.org/3/library/stdtypes.html
  - https://docs.python.org/3/library/functions.html