PYTHON

Data Type and Type Casting

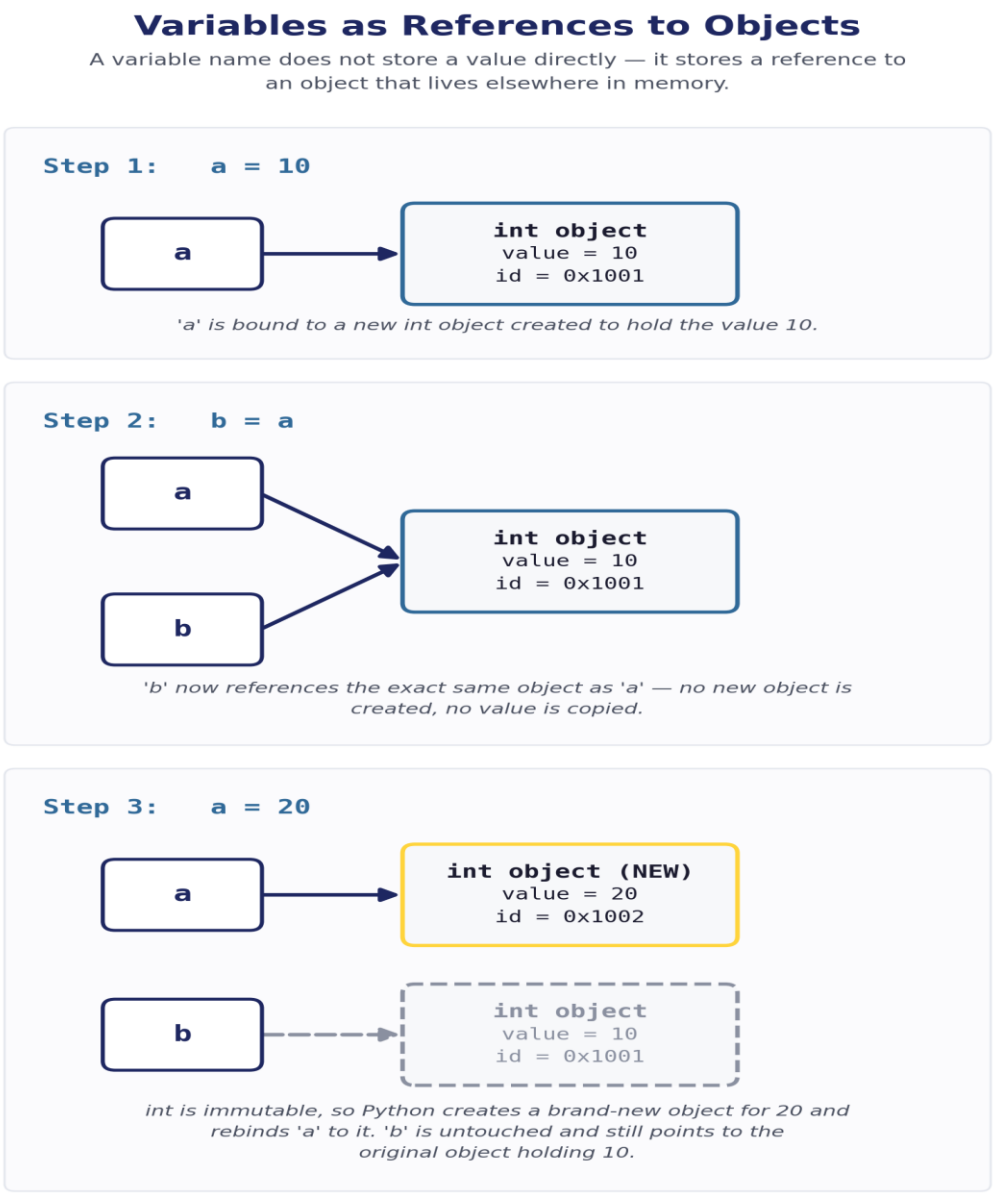
1. **How Data Types Work in Python**

Every piece of data in Python — a number, a string, a list — exists in memory as an object. An object bundles together a type (which determines what operations are valid, e.g. int, str, list), a value, and a unique identity (its memory address, retrievable with the built-in id() function).

A variable, by contrast, is just a name. Writing a = 10 does not create a box called ‘a’ that stores the number 10 inside it. Instead, Python creates an int object holding 10 somewhere in memory, and the name ‘a’ is bound to — that is, it references — that object. This is exactly why Python is described as dynamically typed: the type belongs to the object, not to the variable name, so the same name can later be rebound to an object of a completely different type.

Because a variable is only a reference, assignment never copies data — it only changes what a name points to. Writing b = a makes ‘b’ a second name pointing at the very same object; no new object is created and no value is duplicated. This becomes important the moment a variable is reassigned: since types such as int, float, str, and tuple are immutable, an operation like a = 20 cannot modify the existing object at all. Python must create a brand-new object for 20 and rebind ‘a’ to it, leaving ‘b’ completely unaffected. Mutable types such as list, dict, and set behave differently — their contents can be changed in place, so every name referencing that object sees the update immediately.

The diagram below walks through this exact sequence, step by step.



How data type works in Python

- A variable is a name bound to an object — not a container that holds a value directly
- Assignment (=) binds a name to an object; it never copies the underlying data
- Multiple names can reference the very same object at the same time
- Reassigning a name bound to an immutable object creates a new object — it does not alter the original

- Mutable objects (list, dict, set) can change in place, so every reference to them sees the update
- type() reports the type of the object a name currently refers to; id() reports that object's memory identity
- **Mutable and Immutable**
- **Mutable:** Objects can be changed after creation, like editing a text document. Mutable updates are fast and cheap.
- **Immutable:** Objects cannot be changed once made. Any "change" actually builds a brand-new object in memory. It is safe from accidental changes. Mutable data type locked after its creation. Immutable updates create new memory spaces.

Python Mutability Reference

Data Type	Description	Mutability
list	Ordered sequence of elements	**Mutable**
dict	Collection of key-value pairs	**Mutable**
set	Unordered collection of unique items	**Mutable**
bytearray	Mutable sequence of bytes	**Mutable**
int / **float** / **complex**	All numeric types	**Immutable**
str	Text string sequence	**Immutable**
tuple	Ordered, unchangeable sequence	**Immutable**
bool	Truth values (True, False)	**Immutable**
bytes	Immutable sequence of bytes	**Immutable**
frozenset	Immutable version of a set	**Immutable**

2. Introduction to Data Types

A data type defines the kind of value a variable holds and the operations that can be performed on it. Python is dynamically typed — a variable's type is not declared up front; it is inferred automatically from the value assigned to it, and the same variable can later be reassigned to a value of a completely different type.

You can always check a value's type with the built-in type() function, which is used throughout this document to confirm each example's result.

- Python's built-in types fall into six broad groups: numeric, sequence, mapping, set, boolean, and binary types — plus the special NoneType

- Every value in Python has a type, and every type determines what operations (+, indexing, iteration, etc.) are valid on it

3. Numeric Types

3.1 int (Integer)

Represents whole numbers, positive or negative, with no fixed size limit — unlike many other languages, Python integers grow automatically and never overflow.

Example 1

```
age=25
print(age, type(age))
```

Input: age = 25

```
25 <class 'int'>
```

Example 2 — arbitrarily large integers

```
big_number=10**20
print(big_number, type(big_number))
```

Input: 10 ** 20

```
100000000000000000000 <class 'int'>
```

3.2 float (Floating Point)

Represents real numbers written with a decimal point or in scientific notation. Floats are stored in double precision, so very small rounding errors can appear in arithmetic — this is a property of binary floating-point representation, not a Python bug.

Example 1

```
price = 99.99
print(price, type(price))
```

Input: price = 99.99

99.99 <class 'float'>

Example 2 — a floating-point precision quirk

```
result = 0.1 + 0.2
print(result, type(result))
```

Input: 0.1 + 0.2

0.30000000000000004 <class 'float'>

Note: this happens because 0.1 and 0.2 cannot be represented exactly in binary floating point — a well-known characteristic of float arithmetic in almost every programming language, not specific to Python.

3.3 complex

Represents complex numbers with a real and an imaginary part, written as a + bj. Mainly used in scientific and engineering computation.

Example 1

```
z = 3 + 4j
print(z, type(z), z.real, z.imag)
```

Input: z = 3 + 4j

(3+4j) <class 'complex'> 3.0 4.0

Example 2 — using the complex() constructor

```
z2 = complex(2, -5)
print(z2)
```

Input: complex(2, -5)

(2-5j)

4. Sequence Types

4.1 str (String)

An immutable, ordered sequence of Unicode characters used to represent text. Strings support indexing, slicing, and a large set of built-in methods; being immutable, any “modification” actually creates a new string.

Example 1 — indexing

```
name = "Python"
print(name, type(name), name[0], name[-1])
```

Input: name = "Python"

Python <class 'str'> P n

Example 2 — slicing

```
sentence = "Graduate Studies"
print(sentence[0:8])
```

Input: sentence = "Graduate Studies"

Graduate

4.2 list

An ordered, mutable collection that can hold items of different types together. Elements can be added, removed, or changed after the list is created.

Example 1 — appending an element

```
marks = [85, 90, 78]
marks.append(95)
print(marks, type(marks))
```

Input: marks = [85, 90, 78]

[85, 90, 78, 95] <class 'list'>

Example 2 — mixed data types in one list

```
mixed = [1, "two", 3.0, True]
print(mixed)
```

Input: mixed = [1, "two", 3.0, True]

[1, 'two', 3.0, True]

4.3 tuple

An ordered, immutable collection — once created, its elements cannot be changed, added, or removed. Commonly used for fixed groups of related values, such as coordinates.

Example 1

```
point = (10, 20)
print(point, type(point))
```

Input: point = (10, 20)

(10, 20) <class 'tuple'>

Example 2 — attempting to modify a tuple

```
point = (10, 20)
point[0] = 99
```

Input: point = (10, 20)

TypeError: 'tuple' object does not support item assignment

4.4 range

Represents an immutable sequence of numbers, most often used to control loop iterations. Values are generated lazily, one at a time, rather than stored all at once in memory.

Example 1

```
r = range(1, 6)
print(list(r), type(r))
```

Input: range(1, 6)

[1, 2, 3, 4, 5] <class 'range'>

Example 2 — range with a step

```
r2 = range(0, 20, 5)
print(list(r2))
```

Input: range(0, 20, 5)

[0, 5, 10, 15]

5. Mapping Type

5.1 dict (Dictionary)

A collection of key-value pairs, ordered by insertion since Python 3.7. Keys must be unique and of an immutable type (string, number, tuple); values can be of any type at all.

Example 1

```
student = {"name": "Aman", "age": 24}
print(student, type(student))
```

Input: {"name": "Aman", "age": 24}

{'name': 'Aman', 'age': 24} <class 'dict'>

Example 2 — adding a new key

```
student["grade"] = "A"
print(student["grade"])
```

Input: student["grade"] = "A"

A

6. Set Types

6.1 set

An unordered collection of unique, hashable elements. Duplicate values are automatically discarded, and sets support mathematical operations such as union and intersection.

Example 1 — duplicates are removed automatically

```
numbers = {1, 2, 2, 3, 3, 3}
print(numbers, type(numbers))
```

Input: {1, 2, 2, 3, 3, 3}

{1, 2, 3} <class 'set'>

Example 2 — set operations

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a.union(b), a.intersection(b))
```

Input: a = {1, 2, 3}, b = {2, 3, 4}

{1, 2, 3, 4} {2, 3}

6.2 frozenset

An immutable version of a set. Once created, elements cannot be added or removed — useful when a set-like collection needs to be hashable, such as being used as a dictionary key.

Example 1

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

Input: frozenset([1, 2, 3])

frozenset({1, 2, 3}) <class 'frozenset'>

7. Boolean Type

7.1 bool

Represents one of exactly two values, True or False. bool is technically a subclass of int, where True behaves as 1 and False behaves as 0 in arithmetic. It is the result type of comparisons and logical operations.

Example 1

```
is_valid = 10 > 5
print(is_valid, type(is_valid))
```

Input: 10 > 5

True <class 'bool'>

Example 2 — bool behaving as int

```
print(True + True)
```

Input: True + True

2

8. Binary Types

8.1 bytes

An immutable sequence of bytes (integers from 0 to 255), typically used to represent binary data such as file contents or data sent over a network.

Example 1

```
data = b"hello"
print(data, type(data))
```

Input: b"hello"

b'hello' <class 'bytes'>

8.2 bytearray

A mutable version of bytes — unlike bytes, its individual elements can be changed after creation.

Example 1

```
ba = bytearray(b"hello")
ba[0] = 72
print(ba)
```

Input: bytearray(b"hello"), ba[0] = 72

bytearray(b'Hello')

9. NoneType

None is a special singleton object that represents the absence of a value. It is commonly used as a default argument, a placeholder, or to indicate that a function does not return anything meaningful.

Example 1

```
result = None
print(result, type(result))
```

Input: result = None

None <class 'NoneType'>

10. Type Casting (Type Conversion)

Type casting is the process of converting a value from one data type to another. Python supports two kinds of type conversion: implicit (automatic) and explicit (manual, using built-in functions).

10.1 Implicit Type Conversion

Python automatically converts one data type into another when doing so is safe and loses no information — for example, combining an int and a float in the same expression. This happens internally without any action from the programmer.

Example 1 — int automatically promoted to float

```
a = 5
b = 2.5
c = a + b
print(c, type(c))
```

Input: a = 5, b = 2.5

7.5 <class 'float'>

Example 2 — bool automatically treated as int

```
x = True
y = 10
print(x + y, type(x + y))
```

Input: x = True, y = 10

11 <class 'int'>

10.2 Explicit Type Conversion

Explicit conversion — also called type casting — is performed manually by the programmer using built-in functions such as int(), float(), str(), list(), tuple(), set(), and bool(). It is required whenever Python cannot, or should not, guess the intended conversion on its own.

Example 1 — string to int

```
age_str = "25"
age_int = int(age_str)
print(age_int, type(age_int))
```

Input: age_str = "25"

25 <class 'int'>

Example 2 — int to string, for concatenation

```
score = 90
message = "Score: " + str(score)
print(message)
```

Input: score = 90

Score: 90

Example 3 — float to int (truncates, does not round)

```
value = 9.8
print(int(value))
```

Input: value = 9.8

9

Note: int() on a float always truncates toward zero — it does not round to the nearest whole number. Use round() first if rounding is what you want.

Example 4 — list to tuple and set

```
numbers = [1, 2, 2, 3]
print(tuple(numbers), set(numbers))
```

Input: numbers = [1, 2, 2, 3]

(1, 2, 2, 3) {1, 2, 3}

Example 5 — an invalid conversion

```
value = int("12.5")
```

Input: "12.5"

ValueError: invalid literal for int() with base 10: '12.5'

Note: int() cannot parse a string that contains a decimal point. The correct way to convert such a string is int(float("12.5")), which first converts to 12.5 and then truncates to 12.

10.3 bool() Casting and Truthy / Falsy Values

bool() converts any value to True or False using Python's truthy/falsy rules. The values 0, 0.0, "", [], (), {}, set(), and None are all falsy; virtually every other value is truthy.

Example 1

```
print(bool(0), bool(1), bool(""), bool("hi"), bool([]), bool([1, 2]))
```

Input: 0, 1, "", "hi", [], [1, 2]

False True False True False True

11.1 Summery Table of All Data Type

Sizes below are the immediate (“shallow”) memory footprint reported by sys.getsizeof() for the example value shown, measured on 64-bit CPython 3.12. Actual sizes are

implementation-specific and grow with content — for example, a longer string or a list with more elements takes more bytes — and container types (list, dict, set, tuple) only report their own overhead, not the size of the objects they contain.

Type	Category	Mutable?	Size (bytes)	Example
int	Numeric	No	28	25
float	Numeric	No	24	99.99
complex	Numeric	No	32	3 + 4j
str	Sequence	No	47	"Python"
list	Sequence	Yes	88	[1, 2, 3]
tuple	Sequence	No	64	(1, 2, 3)
range	Sequence	No	48	range(5)
dict	Mapping	Yes	184	{"a": 1}
set	Set	Yes	216	{1, 2, 3}
frozenset	Set	No	216	frozenset([1, 2])
bool	Boolean	No	28	True
bytes	Binary	No	35	b"hi"
bytearray	Binary	Yes	59	bytearray(b"hi")
NoneType	None	No	16	None

11.2 Common Type Casting Functions

Function	Converts To	Example	Result
int()	Integer	int("25")	25

float()	Floating point	float("3.14")	3.14
str()	String	str(90)	"90"
list()	List	list((1, 2, 3))	[1, 2, 3]
tuple()	Tuple	tuple([1, 2, 3])	(1, 2, 3)
set()	Set	set([1, 1, 2])	{1, 2}
bool()	Boolean	bool(0)	False