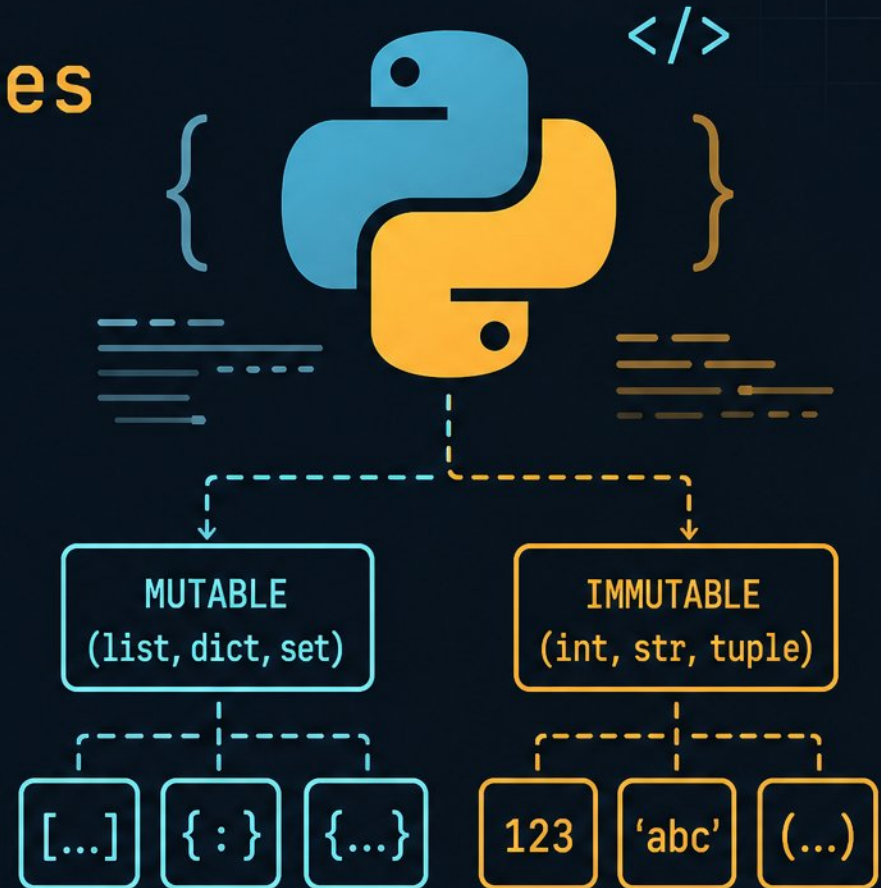


Python

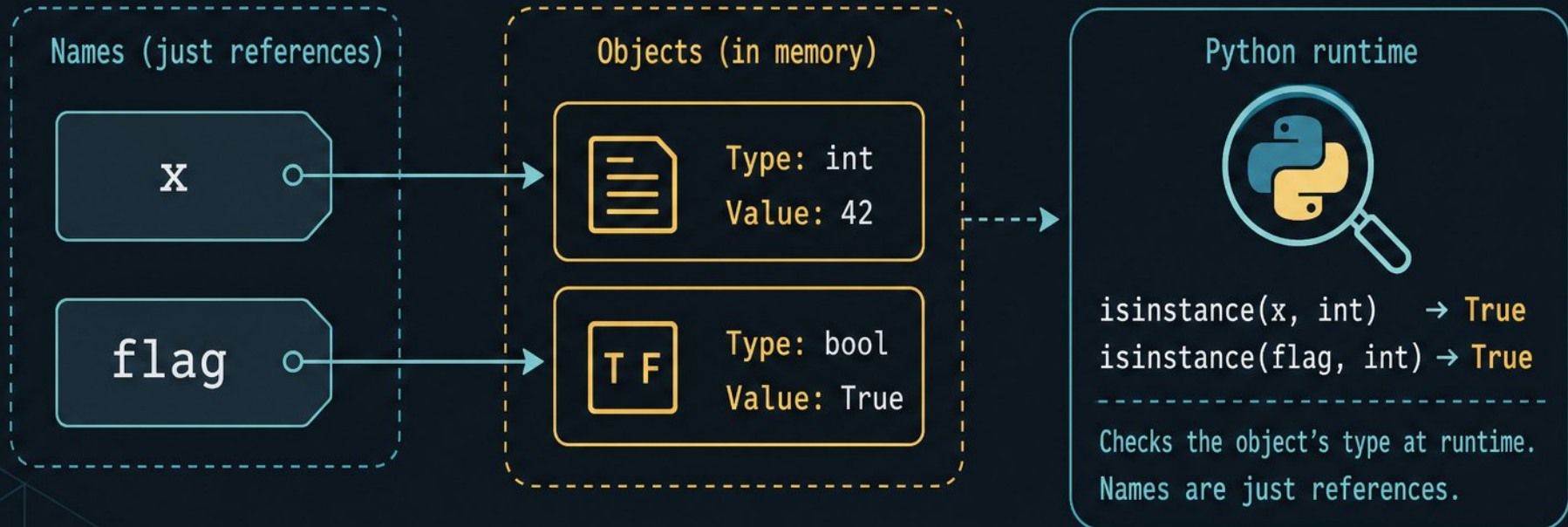
Programming Data Types

- - Data types define valid operations and memory behavior for every value
- - Python is dynamically typed: names are runtime bindings to typed objects
- - Strong typing: Python rejects invalid mixing like 'Hello' + 5 at runtime
- - Two foundational categories: mutable (list, dict, set) and immutable (int, str, tuple)
- - Goal: classify types confidently, choose the right tool, avoid production bugs

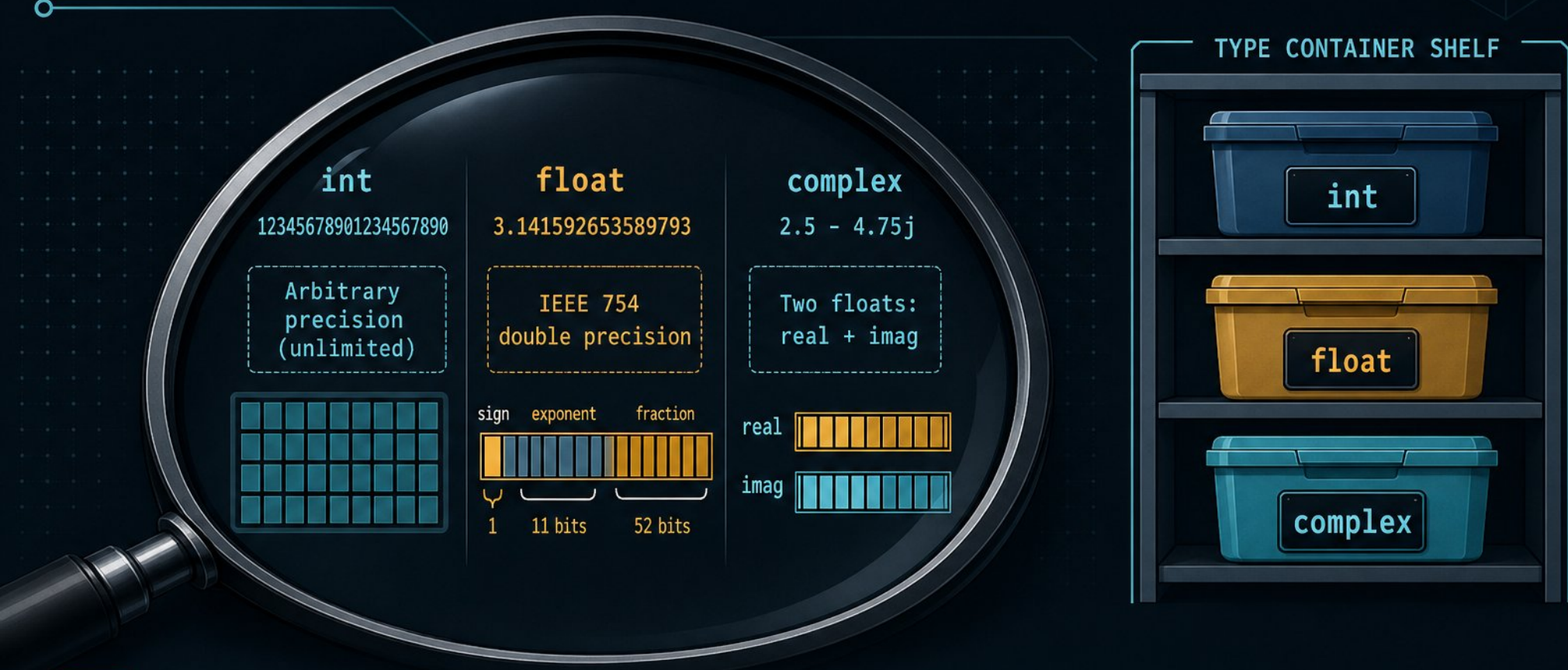


Type Systems and How Python Checks Types

- Static typing: compiler checks types before running, as in Java or C++
- Dynamic typing: Python checks types at runtime, names are just references
- Strong typing: no silent implicit conversions between incompatible types
- Use `isinstance()` for type checks; it respects inheritance like `bool` → `int`
- **Type hints** add optional static checking without changing runtime behavior



Numeric Types: `int`, `float`, and `complex`



123

- `int`: unlimited precision; `float`: double precision; `complex`: real+imag parts

→→

- Mixed-type promotion: `int` → `float` → `complex`; `//` floored, `/` true division

fx()

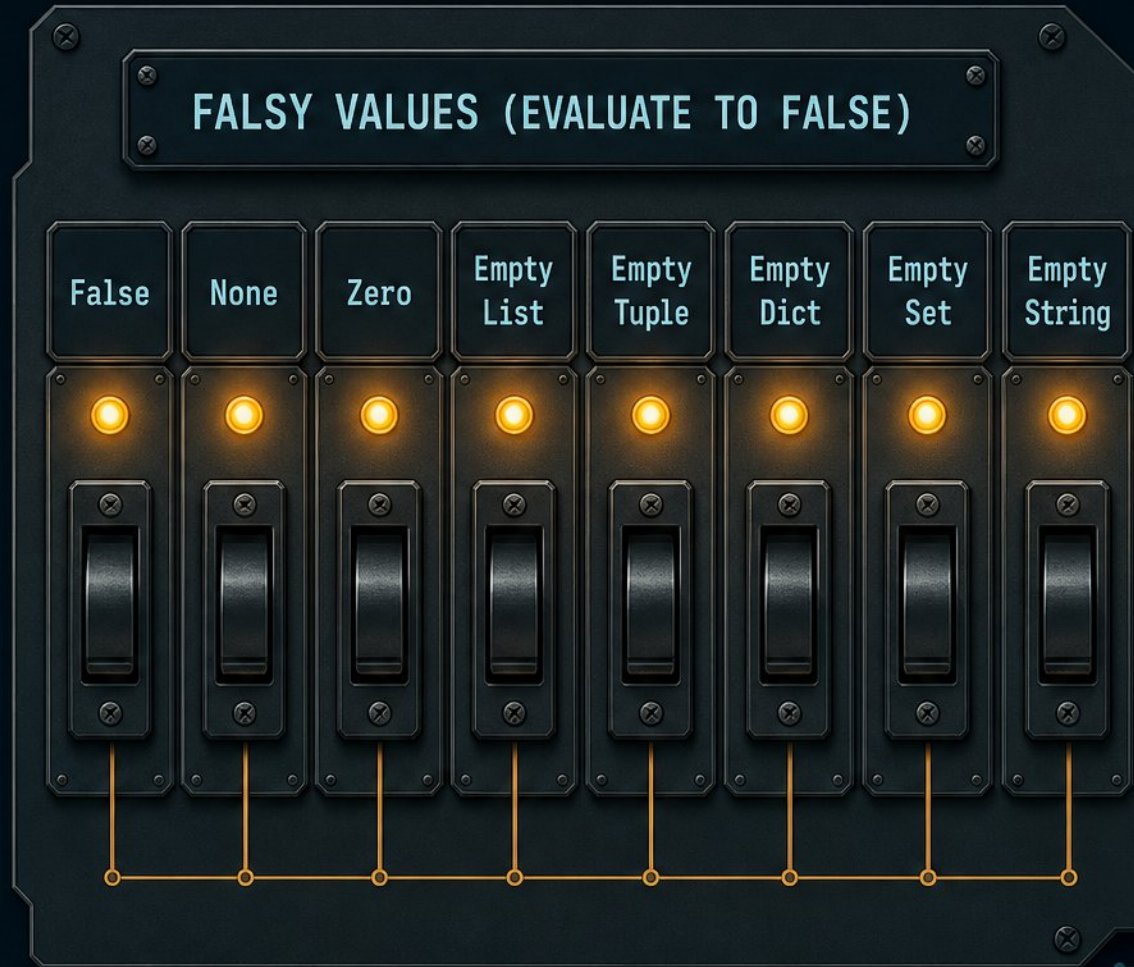
- Built-ins: `abs()`, `round()`, `pow()`, `divmod()`, `floor()`, `ceil()`

!

- **Pitfall:** `0.1+0.2 != 0.3`; use `Decimal` for money

Boolean Type and Truthiness

- `bool` is a subclass of `int`:
`True=1, False=0`
- Falsy values: `False`, `None`, zero, empty containers
- `and/or` return operands;
`not` returns `bool`
- Prefer `and/or/not` over
`&,|,^` for logic



Sequence Types: Strings, Lists, and Tuples

`"hello, world"`

`['a', 'b', 'c']`

`('x', 'y', 'z')`

`str`

`list`

`tuple`

`immutable`

`mutable`

`immutable`

- - `str`: immutable Unicode text; `list`: mutable; `tuple`: immutable
- - Common ops: indexing, slicing, concatenation, repetition, membership, `len()`, `min()`, `max()`
- - Zero-based indexing; negative indices count from end; slice `[start:stop:step]`
- - Choose `list` for mutable collections, `tuple` for fixed, hashable data

Working with Strings in Practice

- - Strings are immutable; every 'modification' creates a new string object
- - Use f-strings for formatting; `split()`, `join()`, `replace()`, `upper()`, `lower()` for transformations
- - Avoid $O(n^2)$ loop concatenation; use `''.join()` for efficiency
- - Unicode text via `str`; bytes for binary data – keep them strictly separate



PITFALL: LOOP CONCATENATION

```
s = s + part
```



BETTER: JOIN SHORTCUT

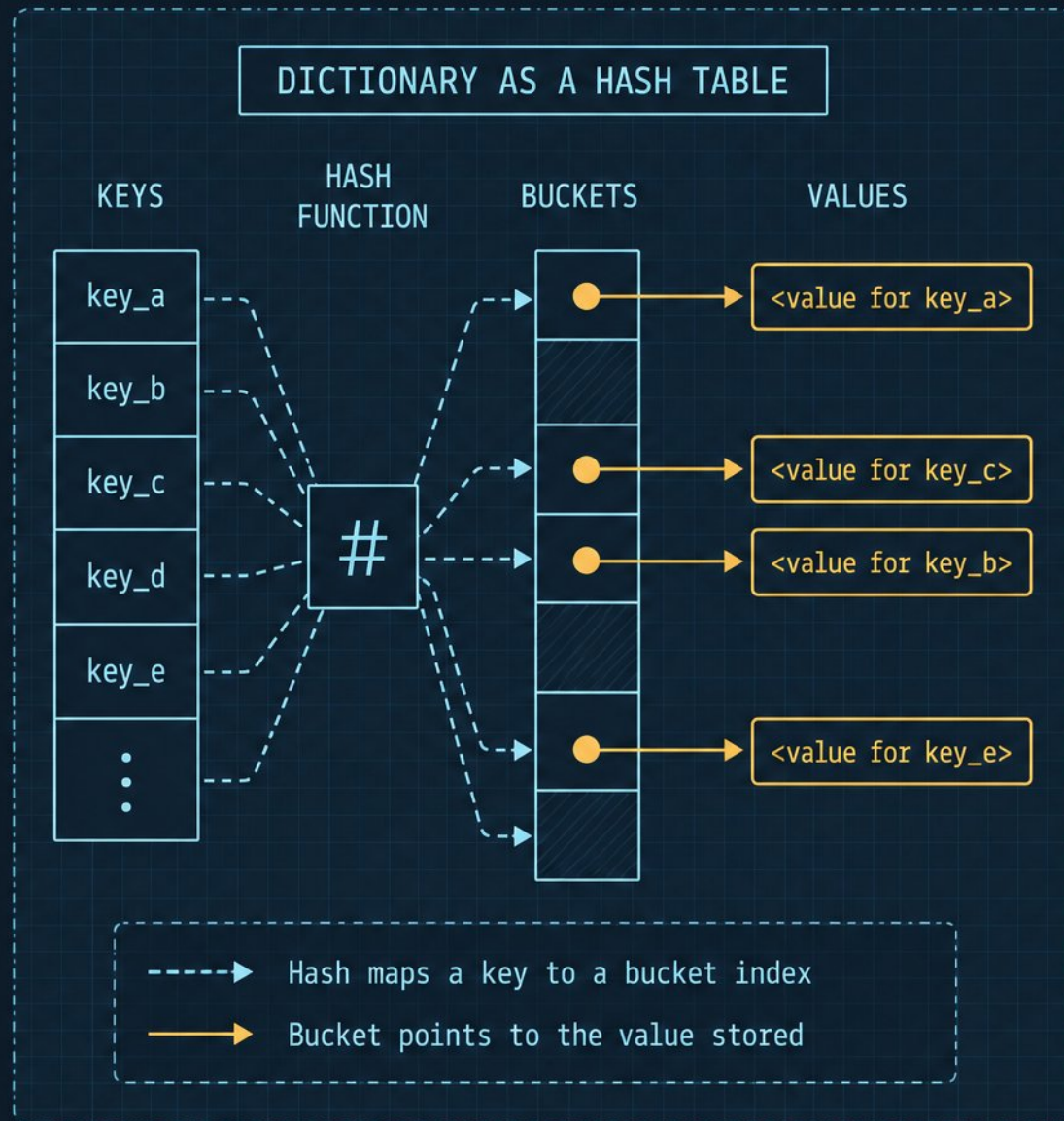
```
result = ''.join(parts)
```



Repeated concatenation copies the entire string each time – use `''.join()` instead.

Mapping Type: Dictionaries

- Dictionaries store unique hashable keys mapped to arbitrary values
- `d[key]` raises `KeyError`;
`d.get(key, default)` handles missing keys safely
- Views and methods: `items()`, `keys()`, `values()`, `update()`, `pop()`, `setdefault()`
- Average $O(1)$ lookup, insert, and delete backed by hash tables
- Iterate with `items()` to avoid repeated lookups in loops





Efficient Dictionary Patterns

1



defaultdict

automatic defaults for grouping and counting

```
from collections import defaultdict
groups = defaultdict(list)
for item in items:
    groups[key(item)].append(item)

# groups[key] creates []
# if key is missing
```

2



Counter

frequency analysis with most_common()

```
from collections import Counter
cnt = Counter(items)
# cnt[key] is 0 if missing
top = cnt.most_common()
top_n = cnt.most_common(n)
```

3



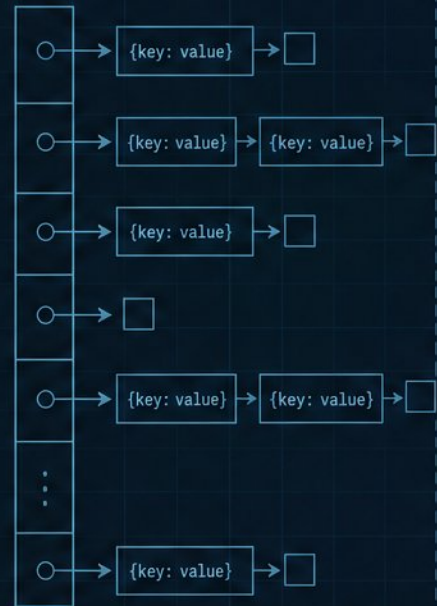
Comprehensions

concise transformations and filtering

```
squares = {x: x*x for x in nums}
evens = {k: v for k, v in data.items()
         if v % 2 == 0}
inverted = {v: k for k, v in data.items()}
```

HASH TABLE

BUCKETS

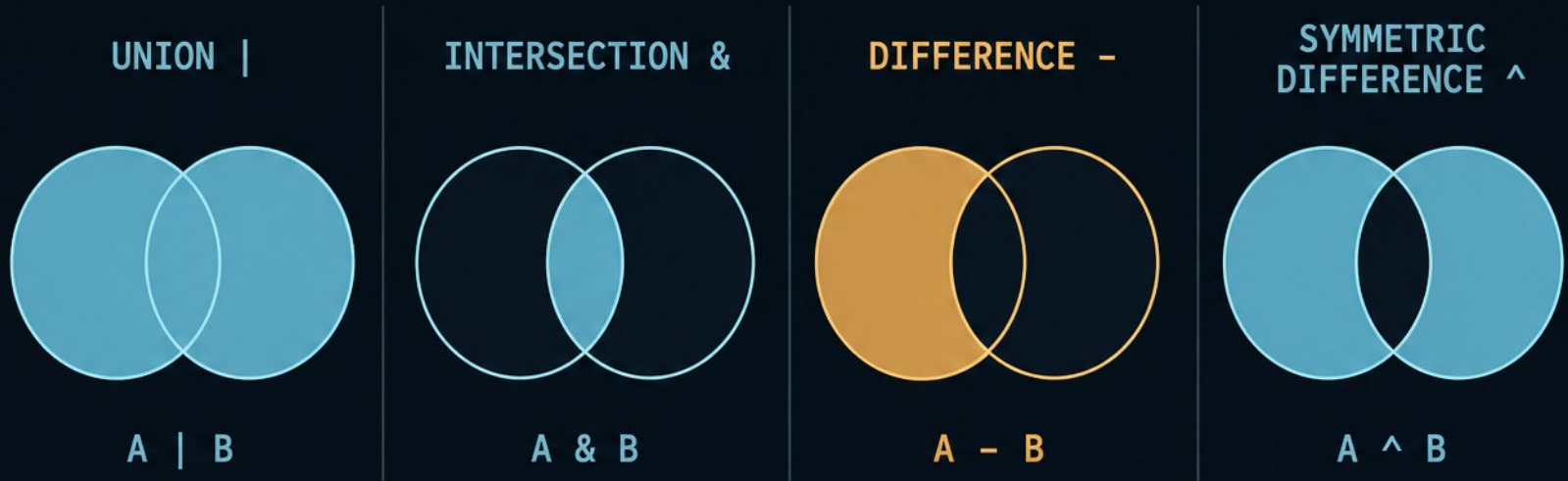


- defaultdict: automatic defaults for grouping and counting
- Counter: frequency analysis with most_common()
- Comprehensions: concise transformations and filtering
- Merge with | and |=; use None defaults, never mutate while iterating

Set Types: Sets and Frozensets



SET ALGEBRA



- Unordered, unique, hashable elements backed by hash tables
- Set algebra: union |, intersection &, difference -, symmetric difference ^
- $O(1)$ membership tests → deduplication, permissions, change detection
- frozenset: immutable version, usable as dict key or set element

NoneType and Representing Absence of Value



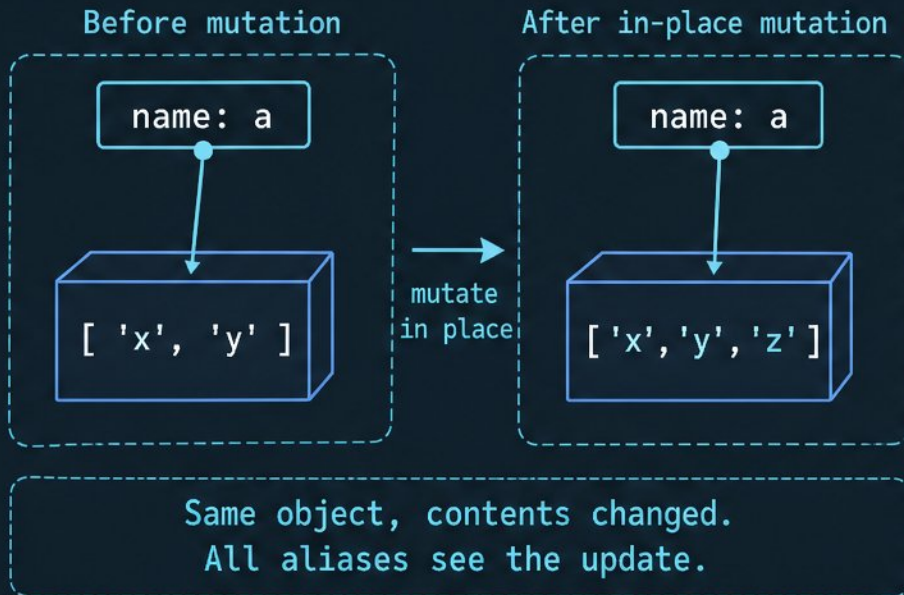
- ▶ **None** is the sole instance of **NoneType**, representing explicit absence of value
- ▶ Always compare with `'is'` or `'is not'`, never with `==`
- ▶ Use **None** as a default argument to avoid mutable default traps
- ▶ Distinguish **None** from falsy values: `0`, empty strings, and empty containers



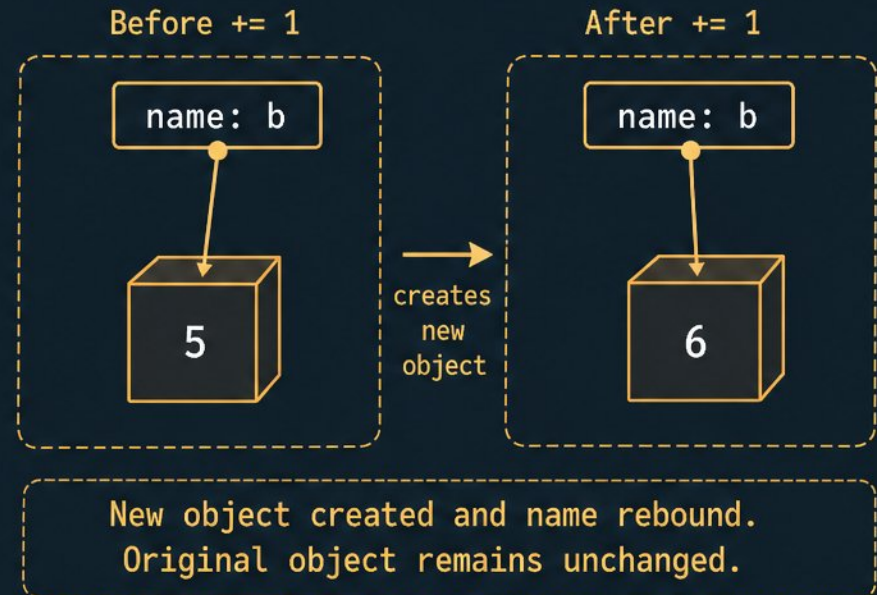
Mutability and Immutability

- - Mutable: `list`, `dict`, `set`, `bytearray`; Immutable: `int`, `float`, `str`, `tuple`, `frozenset`
- - Assignment binds names, never copies; aliases share one mutable object
- - Pass-by-object-reference: mutating arguments affects the caller; rebinding does not
- - `+=` mutates mutable types in place but creates new objects for immutable types

Mutable (list)



Immutable (int)



Name (variable)

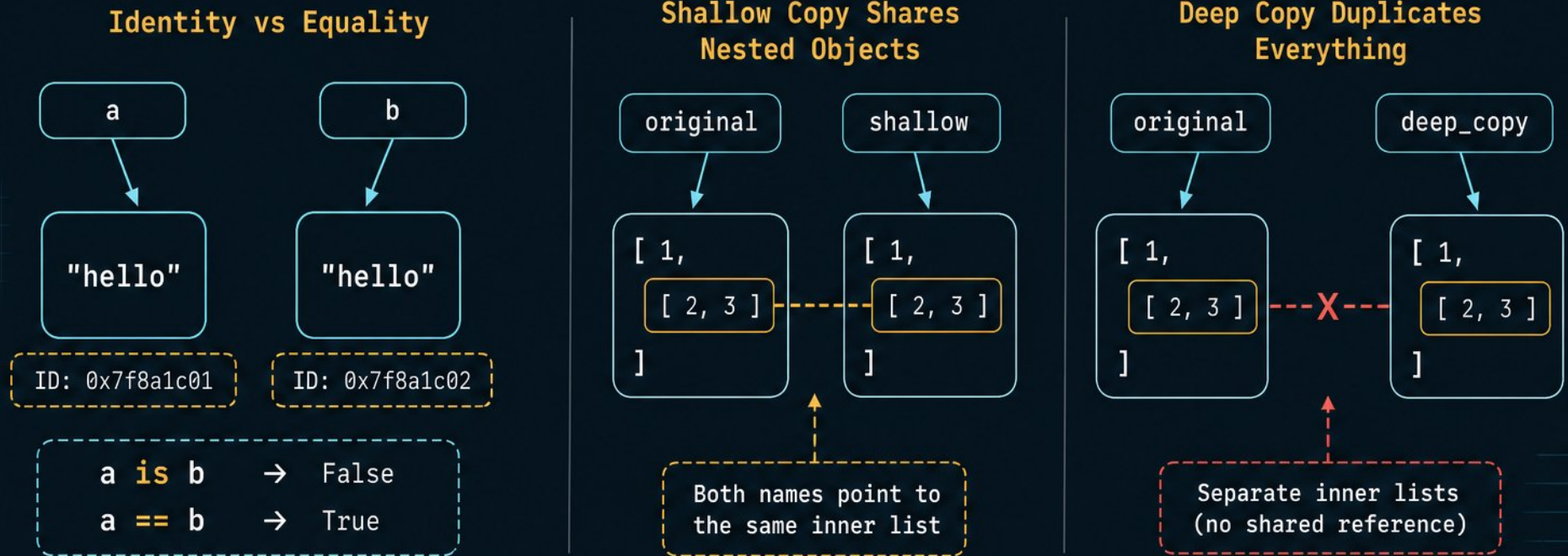


Reference (points to object)



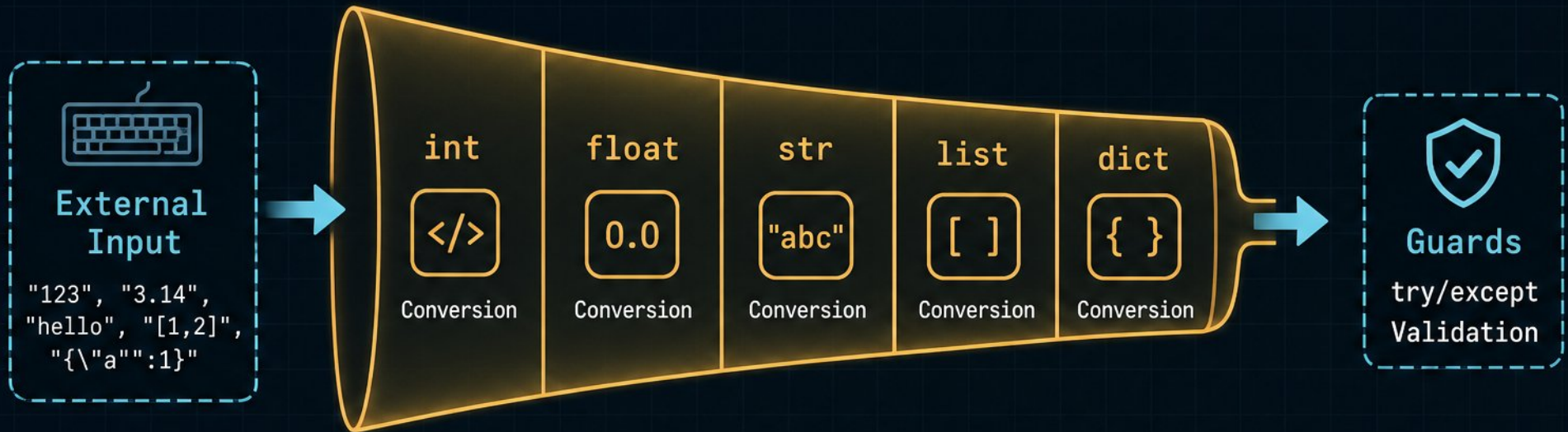
Object

Object Identity, Copying, and Common Pitfalls



- ``is`` checks identity; ``==`` checks value equality
- Never rely on ``is`` for numbers: caching (-5 to 256) makes it unreliable
- Shallow copy shares nested objects; deep copy duplicates everything
- Mutable defaults evaluate once at definition—use ``None`` sentinel instead
- ``+=`` mutates lists in place but creates new objects for tuples and ints

Type Conversion and Practical Strategies

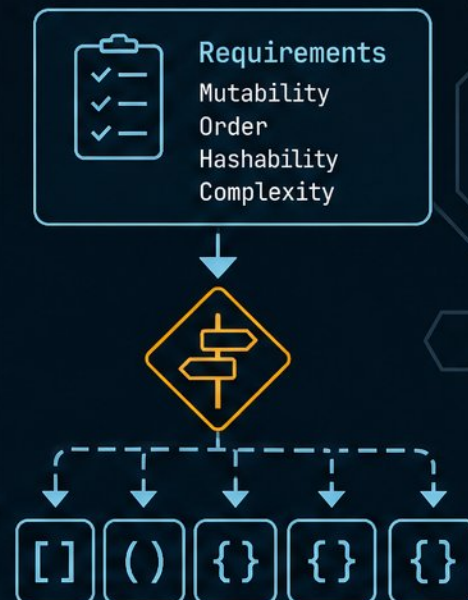


- - Implicit: safe promotion (`int` $\rightarrow$ `float`); explicit: `int()`, `float()`, `str()`
- - Convert at system boundaries: treat conversion as validation
- - Wrap external input in `try/except`; never trust user input
- - `int()` truncates; use `round()` when rounding is intended
- - `.isdigit()` fails for negatives and decimals; prefer `try/except`

Choosing the Right Data Type and Avoiding Production Bugs



Data Type	Mutability	Order	Hashability	Complexity
<code>[]</code> list	Mutable	Ordered	Unhashable	Higher
<code>()</code> tuple	Immutable	Ordered	Hashable	Lower
<code>{}</code> dict	Mutable	Unordered	Unhashable	Higher
<code>{}</code> set	Mutable	Unordered	Hashable	Lower
<code>{}</code> frozenset	Immutable	Unordered	Hashable	Lower



- Choose collections by mutability, order, hashability, and complexity; this is the key design axis



- Use `list` for ordered/mutable, `tuple` for unchanging, `dict` for lookup, `set` for uniqueness, `frozenset` as a dict key



- Add type hints and use `mypy/pyright` as safety nets; they're effectively mandatory in modern production Python



- Validate at system boundaries with `Pydantic` or similar tools to make invalid states unrepresentable

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/94e2728b/public