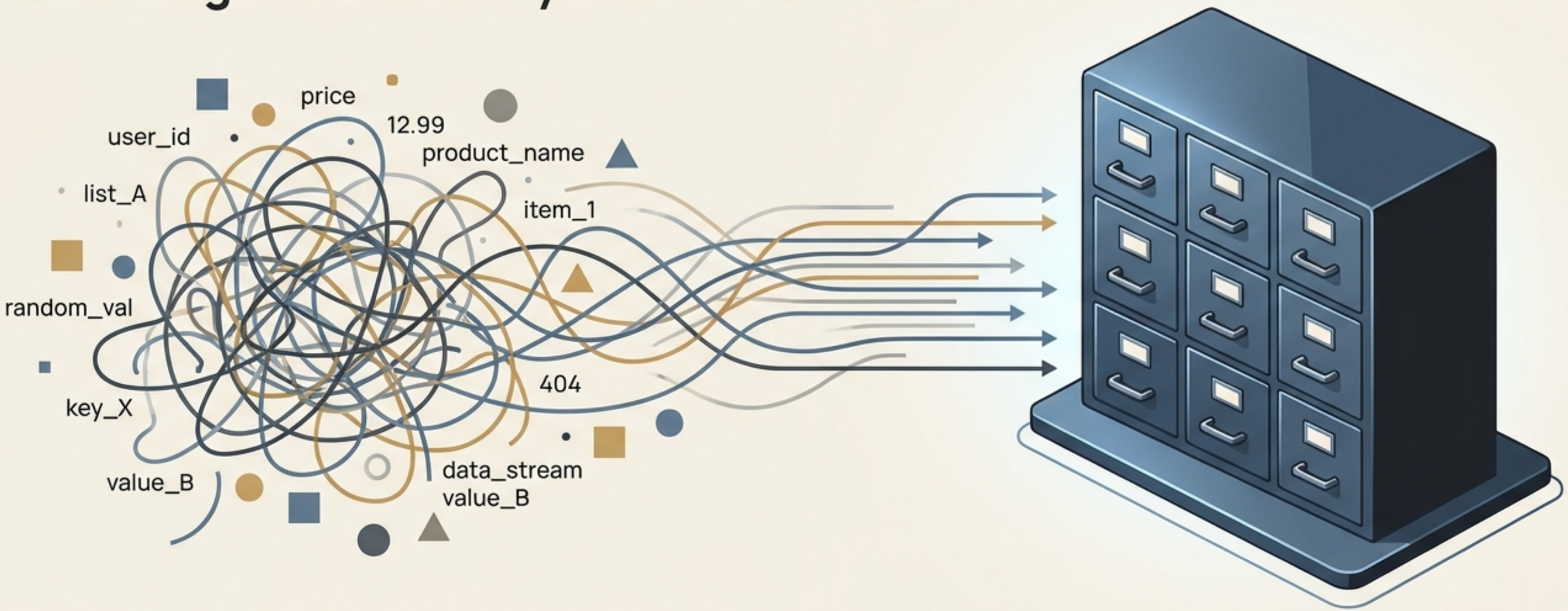
From Data Chaos to Organised Power

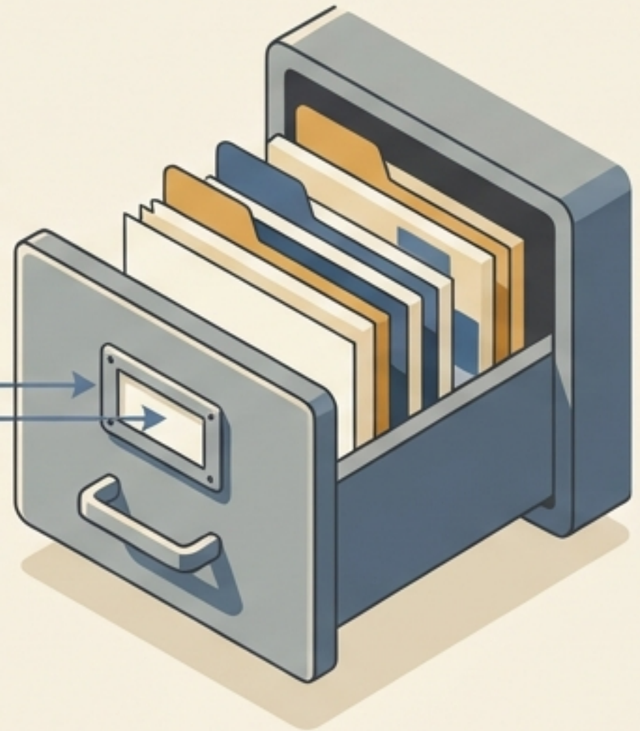
A Strategic Guide to Python Dictionaries



The Digital Filing Cabinet: Understanding Dictionaries

A Python dictionary is a mutable, unordered collection that stores data not in a sequence, but as a set of **key-value pairs.**

Key: It must be unique and provides a direct way to find the contents.



Value: This is the data you want to store and retrieve.

a unique identifier, like a file label

```
# A simple dictionary representing a student
student = {"name": "Yaswant", "age": 22, "city": "Lucknow"}
```

the data stored within

The dictionary container

Three Ways to Construct Your Dictionary

Python offers flexible syntax for creating dictionaries, whether you have data data ready or are starting fresh.

The Classic (Curly Braces)

The most common and readable method.

```
info = {"id": 101,
"role": "admin"}
```

The Constructor (`dict()`)

Useful for creating dictionaries from other data types.

```
data = dict(name="Python",
     level="Advanced")
```

The Blank Slate (Empty)

Start with an empty dictionary and populate it later.

```
d = {}
```

Retrieving Your Data with Precision

Direct Access with Square Brackets

The most direct way to access a value is by using its key. This is fast and efficient.

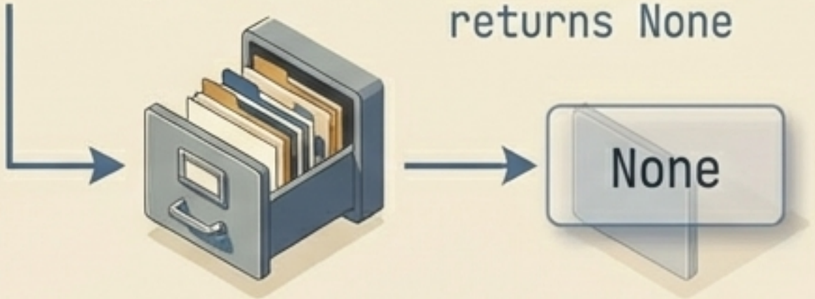
```
student = {"name": "Yaswant", "age": 22}
name = student["name"]  # Returns "Yaswant"
```

> ⚠ This will raise a `KeyError` if the key does not exist.

Safe Access with `.get()`

For situations where a key might be missing, `.get()` is the safer option. It prevents your programme from crashing.

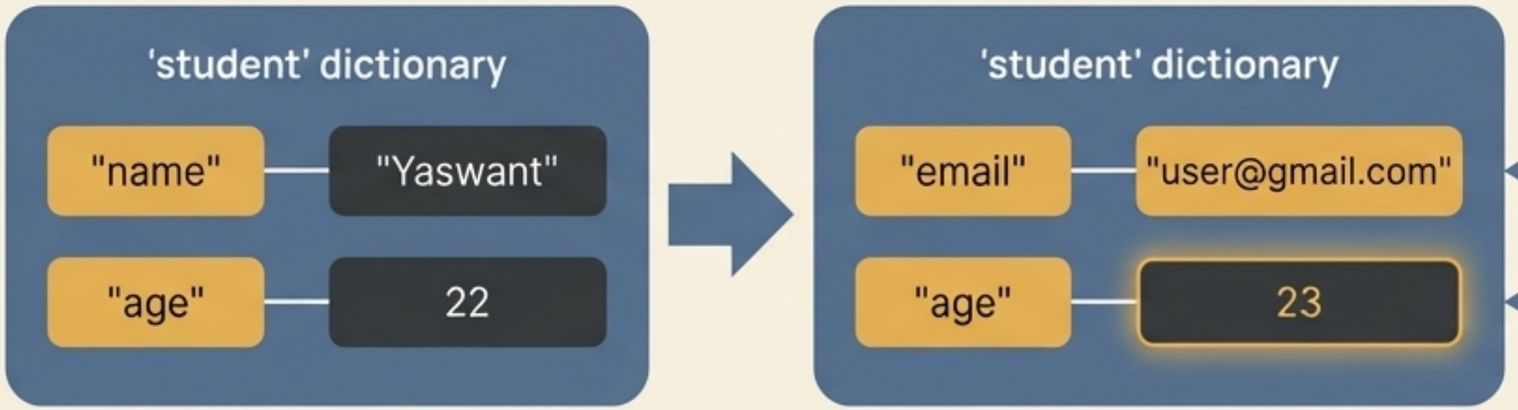
```
age = student.get("age")     # Returns 22
email = student.get("email") # Key is missing,
                             returns None
```



Use `[key]` when you are certain the key exists.
Use `.get()` when the key's presence is uncertain.

The Living Record: Adding and Updating Information

Dictionaries are mutable, meaning their contents can be changed, added to, or removed after creation.



Adding a New Key-Value Pair

To add a new item, simply assign a value to a new key.

```
student["email"] = "user@gmail.com"
```

Updating an Existing Value

To update an item, assign a new value to an existing key. Python simply overwrites the old value.

```
student["age"] = 23
```

Precision Removal: Your Four Options

`pop(key)` — Remove by Key

Removes the item with the specified key and returns its value. Raises a `KeyError` if the key is not found.

```
student.pop("age")
```

`popitem() — Remove the Last Entry

Removes and returns the last key-value pair inserted (for Python 3.7+). Useful for processing items in LIFO (Last-In, First-Out) order.

`del` — The Direct Deletion

A Python keyword (not a method) that removes the item with the specified key. It does not return the value.

```
del student["city"]
```

`clear() — Remove Everything

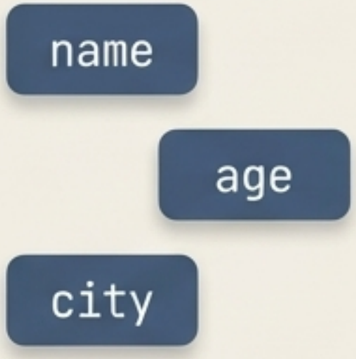
Empties the dictionary of all its items, leaving an empty container `{}`.

```
student.clear()
```

A Systematic Review: Iterating Through Your Dictionary

Dictionaries can be efficiently traversed to work with their keys, values, or both.

Iterating Over Keys



This is the default iteration behaviour.

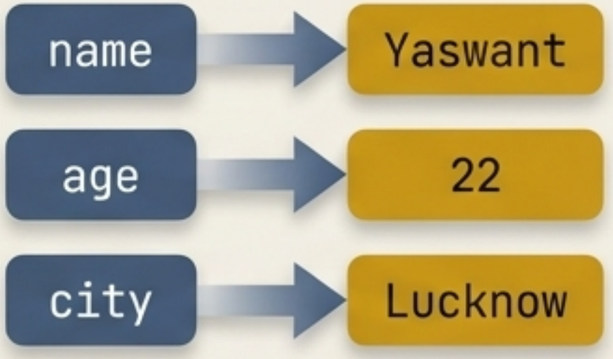
```
for k in student:
    print(k)
```

Iterating Over Values

Use the `.values()` method to access the values directly.

```
for v in student.values():
    print(v)
```

Iterating Over Key-Value Pairs

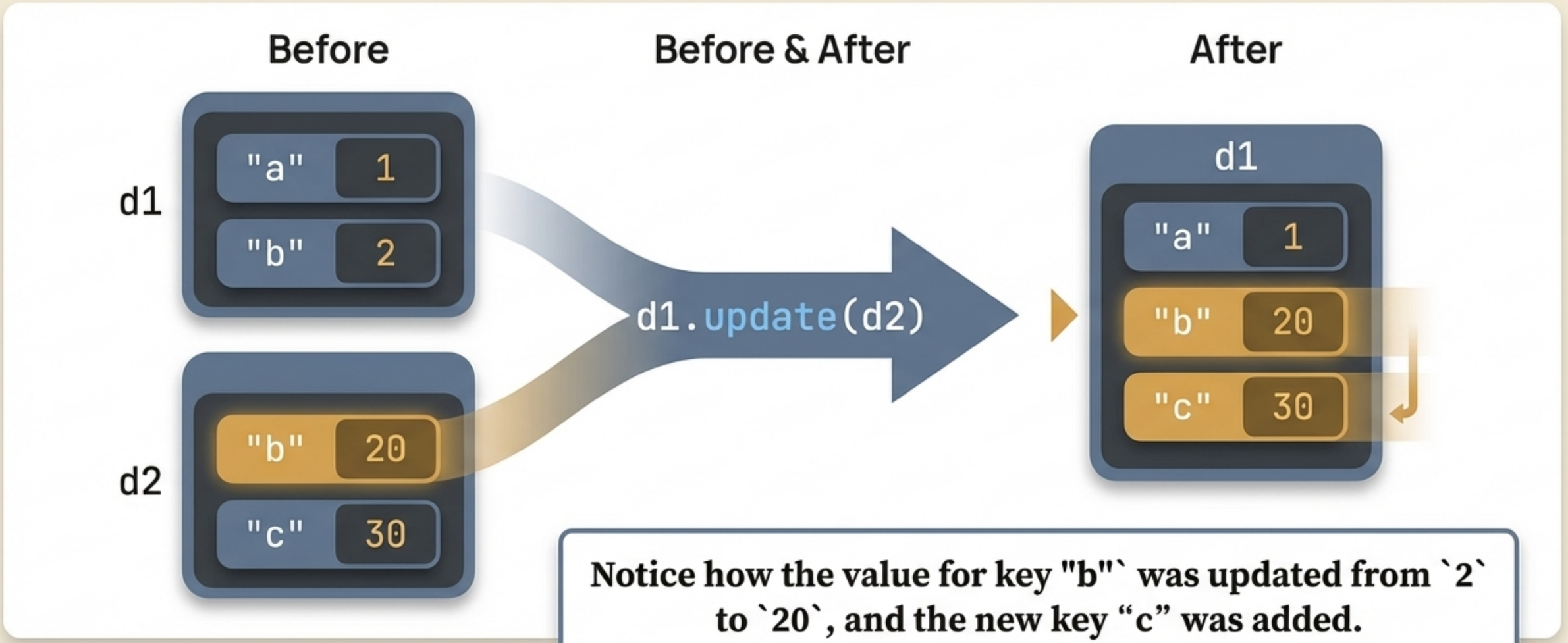


The `.items()` method provides the most complete view, returning a tuple for each pair.

```
for k, v in student.items():
    print(k, v)
```

The `update()` Method: Merging and Overwriting Dictionaries

Concept: The `.update()` method provides a powerful way to merge the contents of one dictionary into
If a key already exists, its value is overwritten.



Notice how the value for key "b"` was updated from `2` to `20`, and the new key “c” was added.

At a Glance: A Quick Reference for Key Methods

Method	Purpose
`.keys()`	Returns a view of all keys in the dictionary.
`.values()`	Returns a view of all values in the dictionary.
`.items()`	Returns a view of all key-value tuple pairs.
`.get(key)`	Safely accesses a value, returning `None` if the key is absent.
`.pop(key)`	Removes an item by its key and returns the value.
`.popitem()`	Removes and returns the last inserted key-value pair.
`.update(other)`	Merges another dictionary, overwriting values for common keys.
`.clear()`	Removes all items from the dictionary.

Essential Rules for Robust Code

Checking for Membership

The `in` keyword checks for the *presence of a key*, not a value. It is a fast and Pythonic way to see if a key exists before trying to access it.

Keys Must Be Immutable

Dictionary keys must be of a data type that cannot be changed, like a string, number, or tuple. Mutable types like lists are not allowed as keys.

Check is for key

key (and fails)

```
"name" in student     # Returns True
"Yaswant" in student # Returns False (checks keys only)

# To check for a value:
"Yaswant" in student.values() # Returns True
```

Check for values

```
d = {(1, 2): "A point"}
```

✅

Valid: A tuple is immutable.

```
d = {[1, 2]: "A list"}
```

❌

TypeError: A list is mutable and cannot be a key.

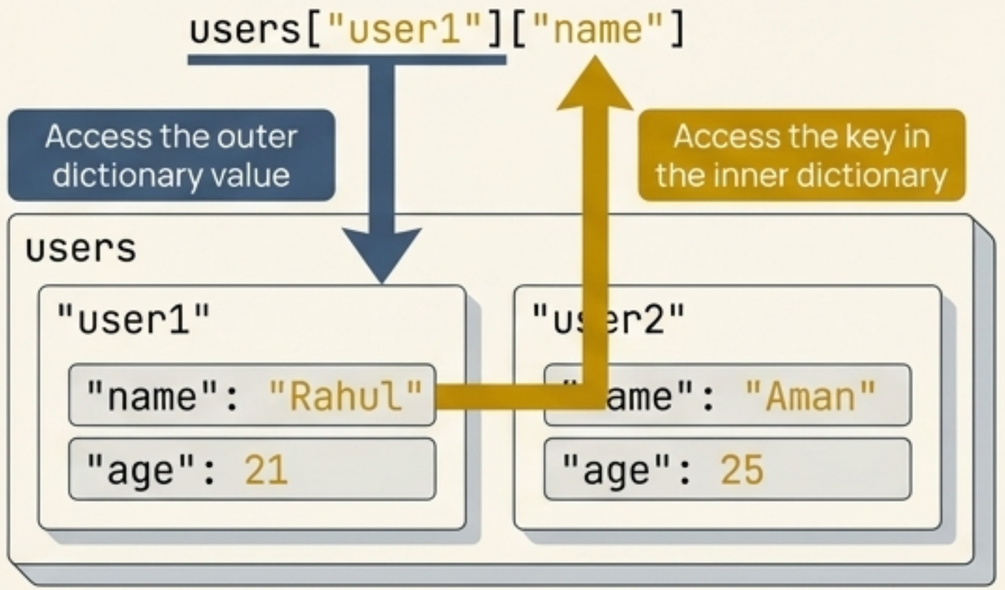
Levelling Up: Building Dictionaries Within Dictionaries

The value associated with a key can be another dictionary. This allows you to create powerful, hierarchical data structures to model complex information.

```
users = {
    "user1": {"name": "Rahul", "age": 21},
    "user2": {"name": "Aman", "age": 25}
}
```

Accessing Nested Data

To access data in a nested dictionary, you chain the keys together using square brackets.



Result:

“Rahul”

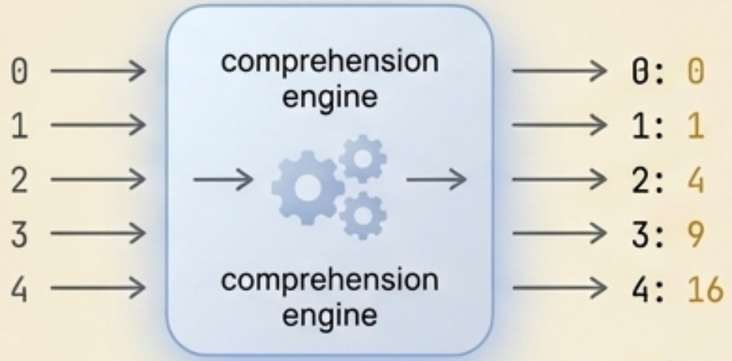
The Pythonic Way: Crafting Dictionaries in a Single Line

Dictionary comprehensions are a compact and readable syntax for creating a dictionary from an existing iterable.

Example 1: Simple Creation

Goal: Create a dictionary of numbers and their squares.

```
squares = {x: x*x for x in range(5)}
```

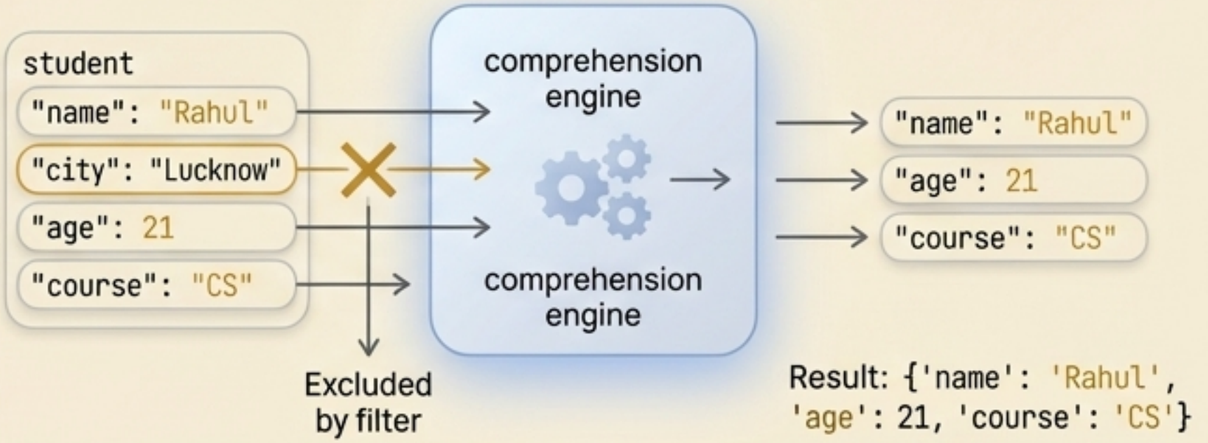


Result: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Example 2: Creation with a Filter

Goal: Create a new dictionary from an existing one, excluding certain values.

```
filtered = {k: v for k, v in student.items() if v != "Lucknow"}
```



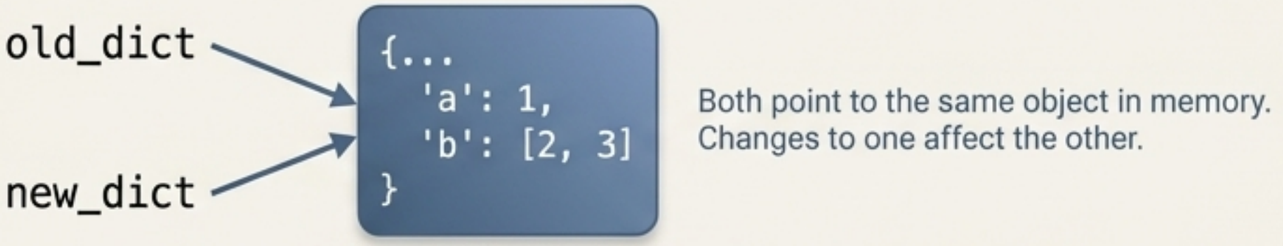
Result: {'name': 'Rahul', 'age': 21, 'course': 'CS'}

Key Insight

This single line combines looping and conditional logic into a highly expressive statement.

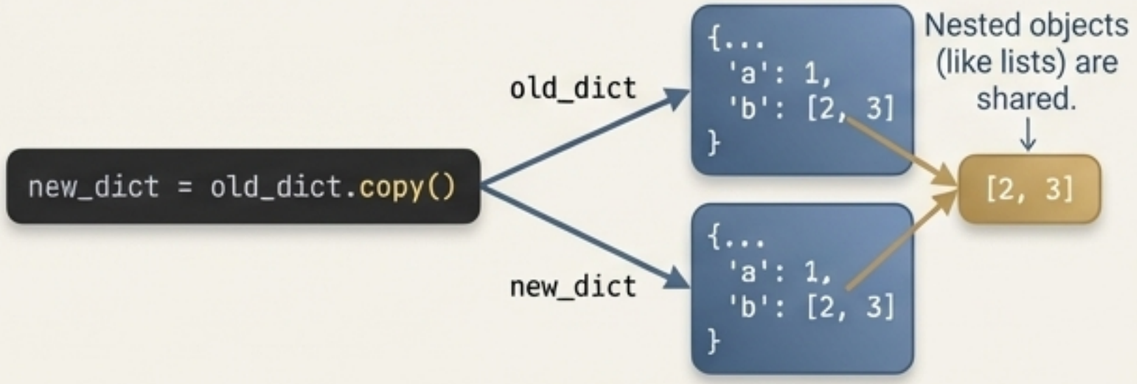
Handling with Care: Creating Independent Copies

The Challenge: Simply assigning a dictionary to a new variable (`new_dict = old_dict`) does not create a copy. Both variables point to the same dictionary in memory. To create a truly separate dictionary, you must explicitly copy it.



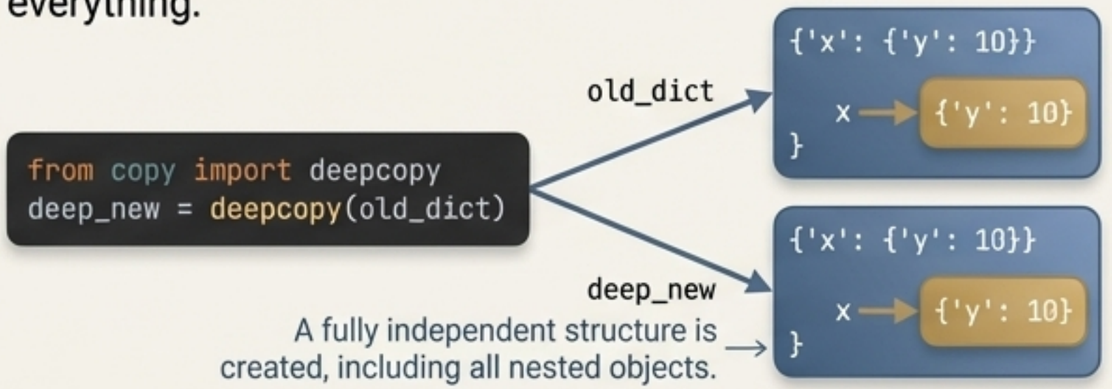
Method 1: Shallow Copy with `.copy()`

Creates a new dictionary object but does not recursively copy nested objects. This is sufficient for simple dictionaries.



Method 2: Deep Copy with `deepcopy()`

For dictionaries containing nested structures (like lists or other dictionaries), `deepcopy` creates a fully independent copy of everything.



From Theory to Practice: Where Dictionaries Shine

The key-value structure of dictionaries makes them the ideal tool for managing structured data in a vast range of common programming tasks.

JSON Data

The structure of JSON objects maps directly to Python dictionaries, making them essential for web development and APIs.

User Profiles

Storing varied pieces of information about a user (name, age, permissions, preferences) is a perfect fit.

Database Mapping

Representing a row from a database table as a dictionary, where column names are keys.

High-Speed Caching

Storing the results of expensive computations with a key for quick retrieval.

API Responses

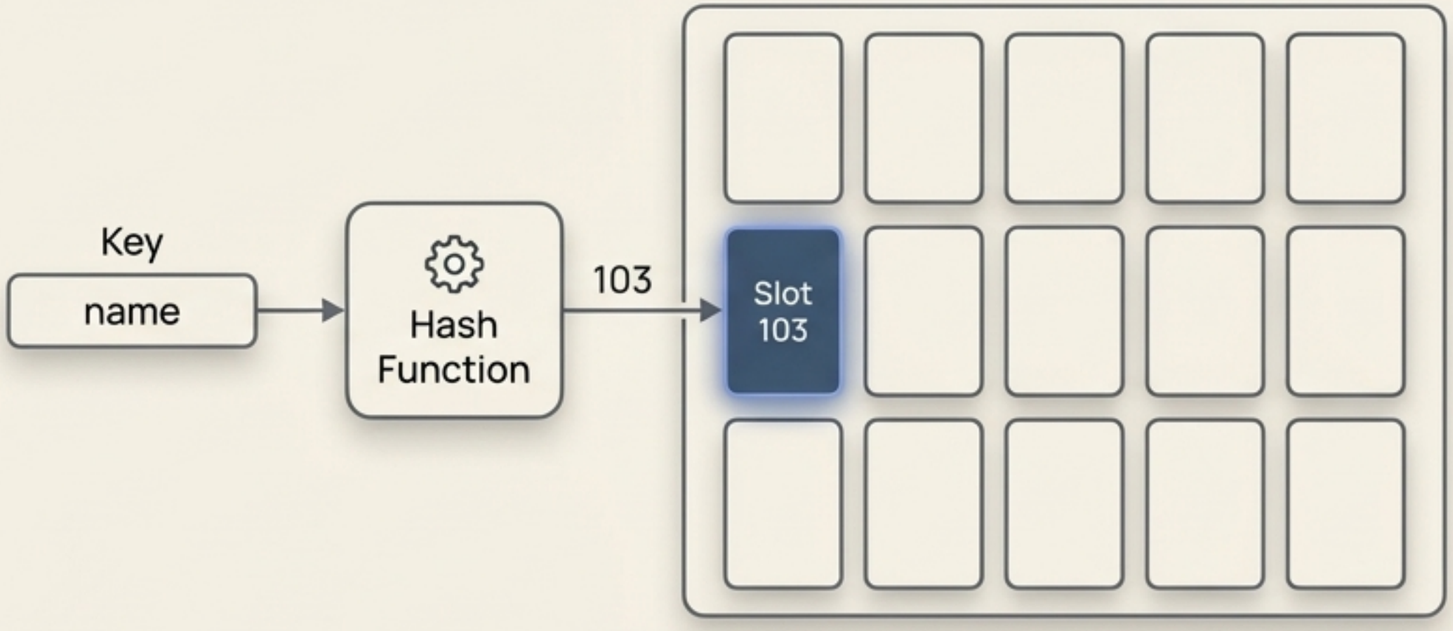
Data returned from external services is almost always structured in a way that is best handled as a dictionary.

Configuration Files

Managing application settings as key-value pairs.

The Cornerstone of High-Performance Python

Dictionaries are more than just a convenience; their underlying design makes them incredibly fast and efficient.



Fast Lookups (Average O(1) Time Complexity)

Because of how dictionaries are implemented (using **hash tables**), retrieving a value by its key does not require searching through the entire collection. On average, the time it takes is **constant**, regardless of whether the dictionary has 10 items or 10 million.

This combination of **flexibility**, **readability**, and **raw speed** is why the dictionary is one of Python's most-used data structures—the foundational tool for bringing organised power to your data.