📘 Chapter 12: Dictionaries – Key-Value Powerhouse

🎯 What You Will Learn

- ✅ Create and access Python dictionaries
- ✅ Use dictionary methods to manipulate data
- ✅ Iterate over keys, values, and items
- ✅ Handle missing keys and default values
- ✅ Apply dictionaries in real-world scenarios

🧠 Why This Concept Matters

Dictionaries are everywhere in Python: configs, APIs, databases, caching, NLP, and even JSON. They're fast (O(1) access), flexible, and critical for modeling structured data. Misusing them (e.g. wrong keys, unsafe lookups) causes hard-to-find bugs.

🔷 Concept Introduction – Dictionaries

Dictionaries in Python are **key-value pairs**, similar to real-world maps or labeled containers. Each key is **unique** and points to a specific value. They're one of the most important data types in Python because they allow **fast lookups**, **structured data modeling**, and **clear associations** between items.

Basic Structure

```
# Syntax: {key: value, key2: value2, ...}
student = {
    "name": "Arjun",
    "age": 18,
    "grade": "A"
}

print(student["name"])  # Access value by key
```

Output:

```
Arjun
```

🧠 Analogy

Think of a dictionary like a **real-life contact list**:

Name	Phone Number
"Alice"	"9876543210"
"Bob"	"9123456789"

In code:

```
contacts = {
    "Alice": "9876543210",
    "Bob": "9123456789"
} # You don't search by position (like a list). You search by key:
print(contacts["Bob"])
```

Output:

```
9123456789
```

💡 Key Rules

- Keys must be **unique** and **immutable** (e.g., strings, numbers, tuples)
- Values can be **any** type — numbers, strings, lists, even other dictionaries
- Dictionaries are **unordered** before Python 3.7 (but insertion-ordered since 3.7+)

❗ Common Pitfall

Trying to access a non-existent key:

```
print(student["email"])  # ❌ KeyError!
```

✔ Use `.get()` to safely access:

```
print(student.get("email", "Not provided"))
```

✅ When to Use Dictionaries

- You want **labelled data** (e.g., a student with fields like name, age)
- You need **quick access** using a known key
- You're working with **structured or nested data**, like JSON

🛠️ Try It Yourself

```
book = {
    "title": "Python Mastery",
    "author": "Naseem",
    "pages": 250
}

print("Book title:", book["title"])
print("Page count:", book.get("pages", 0))
```

🔑 Summary

Concept	Example	Notes
Define	`{"a": 1, "b": 2}`	Key-value pair
Access	`d["a"]`	Raises error if key missing
Safe Access	`d.get("a", default)`	No error if key missing
Modify/Add	`d["c"] = 3`	Adds or updates
Delete	`del d["a"]`	Removes key

🔁 Core 15 Techniques – Python Dictionaries

1. Creating a Dictionary

Feature: Define key-value pairs using curly braces `{}`
Why it matters: It's the foundation of all dictionary operations.

```
car = {"brand": "Toyota", "year": 2020}
print(car)              # Output: {'brand': 'Toyota', 'year': 2020}
```

---

2. Accessing a Value

Feature: Retrieve a value using a key
Why it matters: Lets you read data efficiently in O(1) time.

```
print(car["brand"])     # Output: Toyota
```

---

3. Updating a Value

Feature: Reassign a new value to an existing key
Why it matters: Keeps your data up to date dynamically.

```
car["year"] = 2021
print(car["year"])      # Output: 2021
```

---

4. Adding a New Key

Feature: Add a new entry by assigning to a new key
Why it matters: Expands your dictionary structure on the fly.

```
car["color"] = "red"
print(car)              # Output: {'brand': 'Toyota', 'year': 2021, 'color': 'red'}
```

---

5. Deleting a Key

Feature: Remove a key-value pair using `del`
Why it matters: Frees memory and prevents outdated data usage.

```
del car["year"]
print(car)              # Output: {'brand': 'Toyota', 'color': 'red'}
```

---

6. Checking Key Existence

Feature: Use `in` to verify if a key exists
Why it matters: Prevents runtime errors and improves logic flow.

```
print("brand" in car)   # Output: True
```

---

7. Using .get() Safely

Feature: Access values without raising errors on missing keys
Why it matters: Prevents crashes from unexpected missing keys.

```
print(car.get("owner", "Not found"))   # Output: Not found
```

---

8. Iterating Over Keys

Feature: Loop through each key using a simple for loop
Why it matters: Useful for accessing or displaying dictionary structure.

```
for key in car:
    print(key)          # Output: brand
#         color
```

---

9. Iterating Over Values

Feature: Use `.values()` to loop through only the values
Why it matters: Lets you extract data efficiently without keys.

```
for value in car.values():
    print(value)        # Output: Toyota
#         red
```

---

10. Iterating Over Items

Feature: Use `.items()` to loop through both keys and values
Why it matters: Enables full access to the data in key-value form.

```
for k, v in car.items():
    print(k, v)         # Output:
# brand Toyota
# color red
```

---

11. Using len()

Feature: Count how many key-value pairs exist
Why it matters: Quickly check the size of your dictionary.

```
print(len(car))         # Output: 2
```

---

12. Dictionary Comprehension

Feature: Create dictionaries dynamically with a loop
Why it matters: Powerful tool for transforming or generating data.

```
squares = {x: x**2 for x in range(3)}
print(squares)          # Output: {0: 0, 1: 1, 2: 4}
```

---

13. .pop() for Removing Keys

Feature: Remove a key and get its value in one step
Why it matters: Clean, atomic way to extract and delete.

```
value = car.pop("brand")
print(value)            # Output: Toyota
print(car)              # Output: {'color': 'red'}
```

---

14. .keys(), .values(), .items()

Feature: Access the structure of a dictionary
Why it matters: Useful for inspection, logging, or transformation.

```
print(list(car.keys()))     # Output: ['color']
print(list(car.values()))   # Output: ['red']
print(list(car.items()))    # Output: [('color', 'red')]
```

---

15. Nesting Dictionaries

Feature: Store dictionaries inside dictionaries
Why it matters: Helps model structured data like records, JSON, etc.

```
people = {
  "alice": {"age": 25, "city": "Pune"},
  "bob": {"age": 28, "city": "Delhi"}
}
print(people["alice"]["city"])  # Output: Pune
```

Summary Table – Core Techniques

#	Technique	Code Example	Output
1	Create dict	`{"a": 1}`	`{'a': 1}`
2	Access value	`dict["a"]`	`1`
5	Delete key	`del dict["a"]`	Removes key `'a'`
7	Safe lookup	`dict.get("x", "NA")`	`'NA'`
9	Iterate values	`for v in d.values()`	Prints values
10	Iterate items	`for k,v in d.items()`	Key-value pairs
12	Dict comprehension	`{x: x*x for x in range(3)}`	`{0: 0, 1: 1, 2: 4}`
13	Pop + capture	`v = dict.pop("x")`	Value & removes key

#	Technique	Code Example	Output
14	Keys, values, items	`list(dict.keys())`	Lists of parts
15	Nested dict	`d["a"]["x"]`	Access subkey

Advanced Techniques (Top 5)

1. `setdefault()` – Add if key missing

```
user = {"name": "John"}
user.setdefault("email", "none@example.com")
```

📘 **What it does:**
Checks if `"email"` exists. If yes: returns its value. If no: adds it with default `"none@example.com"` and returns that.

🧠 **Why it matters:**

- Clean one-liner to ensure keys exist.
- Avoids using `if...else` or `try...except`.
- Great for form handling, nested dicts, or APIs.

✅ **Use Case:**

```
data = {}
data.setdefault("score", 0)
data["score"] += 10
```

🧪 **Without setdefault:**

```
if "score" not in data:
    data["score"] = 0
data["score"] += 10
```

---

2. `collections.defaultdict()` – Auto-initialize missing keys

```
from collections import defaultdict
d = defaultdict(int)
d["a"] += 1   # No KeyError!
```

📘 **What it does:**
Creates a dict that auto-generates a default value for missing keys (e.g., 0 for `int`).

🧠 **Why it matters:**

- Avoids manual checks or `setdefault()` when doing counters or grouping.

- Super useful for frequency maps, grouping, or accumulating values.

✅ **Use Case: Word Counter**

```
from collections import defaultdict
counts = defaultdict(int)
for word in ["a", "b", "a"]:
    counts[word] += 1
print(counts)  # {'a': 2, 'b': 1}
```

🔥 **Other modes:**

- `defaultdict(list)` – for grouping
- `defaultdict(dict)` – for nested structures

---

3. Inverting a Dictionary

```
original = {"a": 1, "b": 2}
inv = {v: k for k, v in original.items()}
```

📘 **What it does:**
Flips keys ↔ values.

🧠 **Why it matters:**
Needed in reverse lookups, decoding, or flipping configs.

✅ **Use Case:**

```
status_codes = {"OK": 200, "NOT_FOUND": 404}
reverse_map = {v: k for k, v in status_codes.items()}
print(reverse_map[404])  # Output: NOT_FOUND
```

⚠️ **Watch out:** Values must be unique + hashable for inversion to work.

---

4. Sorting a Dictionary by Value

```
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1]))
```

📘 **What it does:**
Sorts items by their values, not keys.

🧠 **Why it matters:**
Great for rankings, leaderboards, top-n tasks.

✅ **Use Case:**

```
scores = {"Alice": 80, "Bob": 95, "Charlie": 70}
top_scores = dict(sorted(scores.items(), key=lambda x: x[1], reverse=True))
```

🔥 **Bonus:**
Sort by keys using `key=lambda x: x[0]`

---

5. Merging Dictionaries (Python 3.9+)

```
d1 = {"a": 1}
d2 = {"b": 2}
merged = d1 | d2
```

📘 **What it does:**
Combines two dictionaries. `d2` values overwrite if keys overlap.

🧠 **Why it matters:**
Cleaner alternative to `.update()` or unpacking.

✅ **Use Case:**

```
defaults = {"theme": "light", "lang": "en"}
user_settings = {"theme": "dark"}
final = defaults | user_settings
# Output: {'theme': 'dark', 'lang': 'en'}
```

⚠️ **Python 3.8 or older?** Use:

```
merged = {**d1, **d2}
```

Table – Advanced Techniques

#	Technique	Syntax	Use Case
1	`setdefault()`	`dict.setdefault(k, default)`	Safe insert if missing
2	`defaultdict()`	`defaultdict(int)`	Auto-fill missing keys
3	Invert dict	`{v: k for k,v in dict.items()}`	Reverse lookup or decoding
4	Sort by value	`sorted(dict.items(), key=...)`	Leaderboards, scores, rankings
5	Merge (3.9+)	`d1 | d2`	Combine or overwrite dictionary

Mini Quiz (10 Questions)

1. What will `dict["missing"]` do if the key doesn't exist?
2. How can you avoid a `KeyError`?
3. What does `.items()` return?
4. Create a dictionary from two lists.

5. Use dict comprehension to square numbers 1–5.
6. Fix:

   `mydict = {[1,2]: "val"}`
7. What's the output of:

   `len({"a":1, "b":2})?`
8. Can a list be a dictionary key? Why?
9. What's the default return of `get()`?
10. Use `.pop()` to delete a key and capture its value.

Basic Practice (10)

1. **Create a dictionary of student marks.**
   Skill: Dictionary creation
2. **Add a new subject to the dictionary.**
   Skill: Add a new key-value pair
3. **Check if a student exists in the dictionary.**
   Skill: Key existence check using `in`
4. **Use `.get()` to fetch a student's mark safely.**
   Skill: Safe access using `.get()`
5. **Delete a student entry from the dictionary.**
   Skill: `del` or `.pop()`
6. **Iterate over student names.**
   Skill: Looping over keys
7. **Count word frequencies from a string using dictionary.**
   Skill: Frequency counting
8. **Reverse a dictionary (values become keys).**
   Skill: Dictionary comprehension
9. **Sort dictionary values in descending order.**
   Skill: Sorting using `sorted()` and `lambda`
10. **Use dictionary comprehension to map squares of 1–5.**
    Skill: Dict comp

Intermediate Practice (5)

1. **Group names by first letter using `defaultdict`.**
   Use: `defaultdict(list)` for grouping
2. **Merge two user profiles safely.**
   Use: `|` operator (Python 3.9+) or `update()`
3. **Parse key-value pairs from a CSV-like line (e.g., `"name=Ali,age=20"`).**
   Use: `split()` and `dict()` or comprehension

4. **Use nested dictionaries to model a product catalog.**
   Structure: Product → Category → Details
5. **Build a mini leaderboard sorted by score.**
   Use: Sort dict by value (descending)

🔧 Debug Challenges

Challenge 1: Missing key access

```
person = {"name": "Raj", "age": 25}
print(person["gender"])  # ❌ KeyError
# ✅ Fix: Use .get("gender", "Not specified")
```

Challenge 2: Wrong key existence check

```
data = {"score": 10}
if data["points"]:        # ❌ KeyError
    print("Points exist")
# ✅ Fix: Use "points" in data or data.get("points")
```

Challenge 3: Unsafe update on None

```
user = None
user["name"] = "Ali"      # ❌ TypeError: 'NoneType' object is not subscriptable
# ✅ Fix: Make sure `user` is a dict, not None
```

Challenge 4: Mutable default error

```
d2 = {}
d2["key"].append(1)       # ❌ AttributeError: 'int' object has no attribute 'append'
# ✅ Fix: Use setdefault or defaultdict(list)
```

Challenge 5: Collision in inverted dict

```
d = {"a": 1, "b": 1}
inv = {v: k for k, v in d.items()}
print(inv)  # ❌ Only one key will survive
# ✅ Fix: Handle duplicate values before inverting
```

🧠 What You've Learned

You now know how to:

- Create, update, delete, and access key-value data
- Use safe lookup patterns like `.get()`
- Loop through keys, values, and items
- Build smart structures using nesting and comprehension
- Apply advanced tools like `defaultdict`, `setdefault()`, and merging
- Debug common dictionary errors confidently
- Handle real-world data tasks like frequency counting, grouping, and sorting

---

When to Use Dictionaries

Use a dictionary when:

- You need fast lookups (O(1) time) by a label or ID
- You want to map inputs to outputs (e.g., user → email)
- You're modeling structured data (e.g., JSON, APIs, databases)
- You want flexible, expandable data containers

Avoid Dictionaries When:

- You care about ordering before Python 3.7 (older versions)
- You need duplicate keys (not allowed)
- You want sequential operations — use lists for that
- You need fixed keys/fields — consider `dataclass` or `namedtuple`

Compare with Other Types

Type	Best For	Why Choose It
`dict`	Label → value mapping	Fast, flexible, readable key access
`list`	Ordered data, iteration	Numeric index, duplicate elements allowed
`set`	Unique items only	No duplicates, fast membership testing
`tuple`	Fixed-size groups	Immutable, lightweight

> Before you use a dictionary:

- ❓ Do I need named keys?
- ❓ Do I need fast access by label?
- ❓ Will the keys be unique and hashable?

Final Pro Tips

- Use `defaultdict` or `setdefault()` for safe auto-creation.
- Always `.get()` when you're unsure if a key exists.
- Invert only when values are unique.
- Nesting makes your structure powerful—but don't overdo it.

📦 Use This Knowledge To:

- Parse and manipulate API JSON responses
- Build command-line tools with config or routing logic
- Model entities like users, products, tasks, etc.
- Build mini-databases using nested dictionaries
- Implement real-world tasks like leaderboards, groupings, form validation, etc.

🧠 Retention Tip

> Rebuild this entire lesson using:

- `friends`, `books`, `movies`, or `products` as your theme
- Try writing it from scratch with your own data

> > The more you practice dictionary logic with *your* ideas, the faster it'll stick.

Bonus Tools

- `dir(dict)` – explore all dictionary methods
- `help(dict.get)` – deep dive into safe lookups
- Use `json.loads()` to convert JSON to dict

External Docs:

- Python Dict Docs
- RealPython on Dictionaries