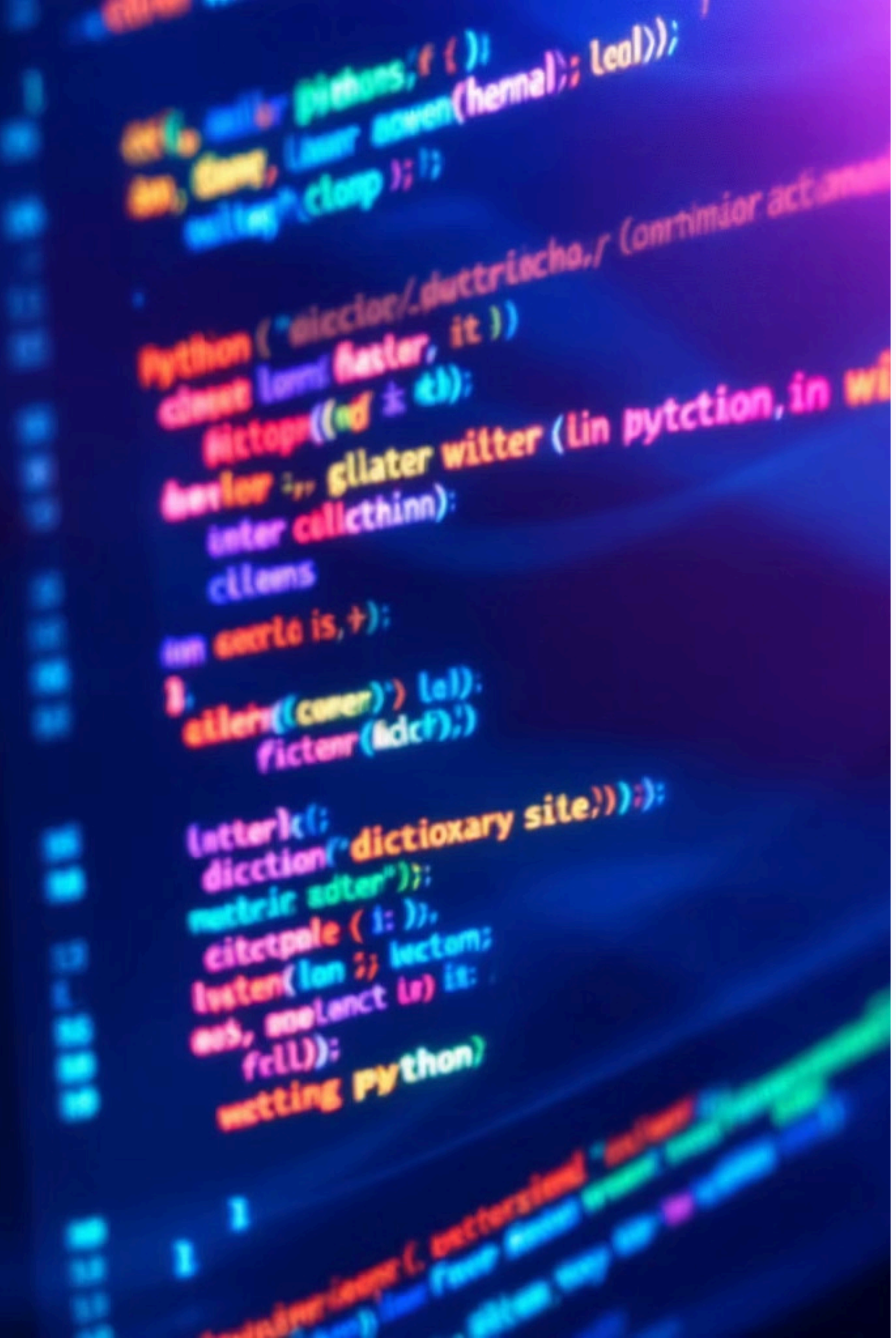


BEGINNER GUIDE PYTHON FUNDAMENTALS

Python Dictionary Basics for Beginners

Master one of Python's most powerful and versatile data structures — the dictionary. This guide walks you through everything you need to know, from creating your first dictionary to manipulating data like a pro. Whether you're just starting your Python journey or brushing up on fundamentals, this presentation covers concepts, real-world examples, and hands-on exercises to solidify your understanding.

What is a Python Dictionary?

A **dictionary** in Python is a built-in data structure that stores data as **key-value pairs**. Think of it exactly like a real-world dictionary: every word (the *key*) maps to its definition (the *value*). In Python, this concept is generalized so that keys and values can represent virtually any kind of association — names to phone numbers, product codes to prices, or country names to their capitals.

Core Characteristics

- **Key-Value Pairs:** Every entry consists of a unique key paired with a value. You look up values by referencing their key, not by a numeric index like you would in a list.
- **Keys Must Be Immutable:** Keys can be strings, numbers, or tuples — but never lists or other dictionaries. This immutability requirement ensures that keys remain hashable and can be quickly looked up internally.
- **Values Can Be Anything:** Values have no restrictions. They can be strings, integers, floats, lists, other dictionaries, or even custom objects.
- **Mutable Structure:** You can freely add new key-value pairs, modify existing values, or remove entries after the dictionary has been created.
- **Unordered (Historically):** Prior to Python 3.7, dictionaries did not guarantee insertion order. From Python 3.7+, dictionaries maintain the order in which items were inserted, though they are still conceptually thought of as unordered mappings.

Quick Example

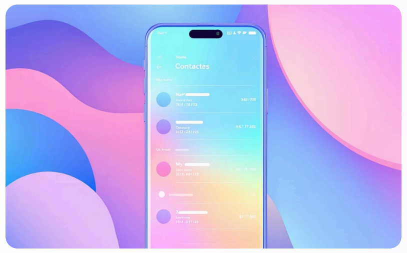
```
fruits = {
  "apple": "red",
  "banana": "yellow",
  "grape": "purple"
}

print(fruits["apple"])
# Output: red

print(fruits["banana"])
# Output: yellow
```

In this example, fruit names serve as **keys** and their colors are the **values**. Accessing a value is as simple as passing the key inside square brackets.

💡 **Key takeaway:** Dictionaries give you instant access to data through meaningful labels rather than arbitrary index numbers. This makes your code more readable and self-documenting.

Why Use Dictionaries?

Lists are great when you have ordered collections, but what happens when you need to associate one piece of data with another? That's where dictionaries shine. They provide a natural, intuitive way to model real-world relationships between data points.



Meaningful Access

Imagine storing phone numbers. With a list, you'd need to remember that index 0 is Mom and index 1 is Dad. With a dictionary like `phones = {"Mom": "555-1234", "Dad": "555-5678"}`, you simply ask for `phones["Mom"]`. No memorization of positions required — the key itself tells you what the value represents.

Lightning-Fast Lookups

Under the hood, Python dictionaries use a **hash table**, which means looking up a value by its key takes **O(1) average time** — constant time regardless of how large the dictionary grows. Compare this to a list where searching for an item takes O(n) time. For large datasets, this speed difference is enormous.

Real-World Modeling

Dictionaries naturally represent relationships found in everyday data: **products & prices**, **students & grades**, **countries & capitals**, **usernames & passwords**, and **configuration settings**. Any time you think "this maps to that," a dictionary is the right tool.

Dictionary vs. List — When to Choose What

Feature	List	Dictionary
Access by	Numeric index (0, 1, 2…)	Meaningful key ("name", "age")
Lookup speed	O(n) for searching	O(1) average for key lookup
Best for	Ordered sequences of items	Labeled, associative data
Duplicates	Allows duplicate values	Keys must be unique

Creating a Dictionary in Python

There are several ways to create a dictionary in Python, each suited to different situations. Let's explore the most common approaches with detailed examples.

Method 1: Curly Braces `{}`

The most common and readable way — use curly braces with `key: value` pairs separated by commas:

```
# Basic dictionary creation
student = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science",
    "gpa": 3.8
}

print(student)
# {'name': 'Alice', 'age': 21,
#  'major': 'Computer Science',
#  'gpa': 3.8}
```

Method 2: `dict()` Constructor

Use the built-in `dict()` function with keyword arguments. Note that keys become strings automatically:

```
# Using dict() constructor
student = dict(
    name="Alice",
    age=21,
    major="Computer Science",
    gpa=3.8
)

print(student["name"])
# Output: Alice
```

Method 3: From a List of Tuples

Convert a list of `(key, value)` tuples into a dictionary. Useful when building dictionaries dynamically:

```
# From a list of tuples
pairs = [
    ("apple", 1.50),
    ("banana", 0.75),
    ("cherry", 3.00)
]
prices = dict(pairs)

print(prices)
# {'apple': 1.5, 'banana': 0.75,
#  'cherry': 3.0}
```

Method 4: Empty Dictionary

Start with an empty dictionary and add items as needed:

```
# Empty dictionary
inventory = {}

# Adding items one by one
inventory["pencils"] = 50
inventory["notebooks"] = 30
inventory["erasers"] = 100

print(inventory)
# {'pencils': 50, 'notebooks': 30,
#  'erasers': 100}
```

⚠️ **Common mistake:** Using `set_name = {"apple", "banana"}` creates a *set*, not a dictionary! You need the colon syntax `{"apple": 1, "banana": 2}` to create a dictionary. An empty `{}` creates an empty dictionary, while `set()` creates an empty set.

CORE OPERATIONS

Accessing, Adding & Modifying Values

Once you've created a dictionary, the real power lies in how you interact with it. Let's explore the essential operations you'll use every day when working with dictionaries.

1

Accessing Values

Use square bracket notation `dict[key]` to retrieve a value. If the key doesn't exist, Python raises a `KeyError`. To avoid errors, use the `.get()` method, which returns `None` (or a default value you specify) when the key is missing.

2

Adding New Pairs

Simply assign a value to a new key: `dict["new_key"] = value`. If the key already exists, the old value is overwritten. You can also use `.update()` to add multiple pairs at once from another dictionary or iterable.

3

Modifying Existing Values

Reassign a key to a new value: `dict["key"] = new_value`. The syntax for adding and modifying is identical — Python determines which operation to perform based on whether the key already exists.

Code Examples in Action

```
student = {"name": "Alice", "age": 21, "major": "CS"}

# 1. Accessing values
print(student["name"])        # Output: Alice
print(student.get("gpa"))     # Output: None (no error!)
print(student.get("gpa", 0.0))  # Output: 0.0 (custom default)

# 2. Adding a new key-value pair
student["gpa"] = 3.8
print(student)
# {'name': 'Alice', 'age': 21, 'major': 'CS', 'gpa': 3.8}

# 3. Modifying an existing value
student["age"] = 22
print(student["age"])        # Output: 22

# 4. Adding multiple pairs at once
student.update({"email": "alice@uni.edu", "year": "Senior"})
print(student)
# {'name': 'Alice', 'age': 22, 'major': 'CS', 'gpa': 3.8,
#  'email': 'alice@uni.edu', 'year': 'Senior'}
```

💡 Pro Tip

Always prefer `.get()` over bracket notation when you're unsure if a key exists. It prevents your program from crashing with a `KeyError`.

🔑 Remember

Dictionary keys are **case-sensitive**. `"Name"` and `"name"` are treated as two completely different keys.

Removing Items & Useful Methods

Python provides multiple ways to remove items from a dictionary, plus a rich set of built-in methods that make working with dictionaries efficient and elegant. Understanding these methods is essential for writing clean, Pythonic code.

Removing Items

```
# Setup
car = {
    "brand": "Toyota",
    "model": "Camry",
    "year": 2023,
    "color": "blue"
}

# del — removes by key
del car["color"]

# .pop() — removes and returns
model = car.pop("model")
print(model)  # Output: Camry

# .pop() with default (no error)
x = car.pop("price", "N/A")
print(x)  # Output: N/A

# .popitem() — removes last pair
last = car.popitem()
print(last)  # ('year', 2023)

# .clear() — empties entire dict
car.clear()
print(car)  # Output: {}
```

Essential Methods

```
person = {
    "name": "Bob",
    "age": 30,
    "city": "NYC"
}

# .keys() — all keys
print(person.keys())
# dict_keys(['name', 'age', 'city'])

# .values() — all values
print(person.values())
# dict_values(['Bob', 30, 'NYC'])

# .items() — all key-value tuples
print(person.items())
# dict_items([('name', 'Bob'),
#   ('age', 30), ('city', 'NYC')])

# len() — number of pairs
print(len(person))  # Output: 3

# "in" — check if key exists
print("name" in person)  # True
print("email" in person) # False

# .copy() — shallow copy
backup = person.copy()
```

Quick Reference: Dictionary Methods at a Glance

Method	Description	Example
.get(key, default)	Returns value or default if key missing	d.get("age", 0)
.keys()	Returns a view of all keys	d.keys()
.values()	Returns a view of all values	d.values()
.items()	Returns key-value pairs as tuples	d.items()
.pop(key)	Removes key and returns its value	d.pop("name")
.update(other)	Merges another dict into this one	d.update({"x": 1})
.setdefault(key, val)	Returns value if key exists, else sets it	d.setdefault("age", 25)
.clear()	Removes all items from dictionary	d.clear()

ITERATION

Looping Through Dictionaries

One of the most common tasks with dictionaries is iterating through their contents. Python provides elegant ways to loop through keys, values, or both simultaneously. Mastering these patterns will make your code cleaner and more efficient.

Four Ways to Loop

```
scores = {"Alice": 95, "Bob": 87, "Charlie": 92, "Diana": 98}

# 1. Loop through keys (default behavior)
for student in scores:
 print(student) # Alice, Bob, Charlie, Diana

# 2. Loop through values only
for score in scores.values():
 print(score) # 95, 87, 92, 98

# 3. Loop through key-value pairs (most common!)
for student, score in scores.items():
 print(f"{student} scored {score}")
 # Alice scored 95
 # Bob scored 87 ...etc.

# 4. Check membership while looping
for student, score in scores.items():
 if score >= 90:
 print(f"
```

Nested Dictionaries & Real-World Patterns

In real-world applications, data is rarely flat. **Nested dictionaries** — dictionaries inside dictionaries — allow you to model complex, hierarchical data structures. This is one of the most practical and commonly used patterns in Python programming, especially when working with APIs, JSON data, and configuration files.

Building a Nested Dictionary

```
# A classroom of students with detailed records
classroom = {
    "Alice": {
        "age": 21,
        "grades": {"math": 95, "english": 88, "science": 92},
        "active": True
    },
    "Bob": {
        "age": 22,
        "grades": {"math": 78, "english": 85, "science": 80},
        "active": True
    },
    "Charlie": {
        "age": 20,
        "grades": {"math": 90, "english": 92, "science": 88},
        "active": False
    }
}

# Accessing nested values — chain the keys!
print(classroom["Alice"]["grades"]["math"])    # Output: 95

# Modifying a nested value
classroom["Bob"]["grades"]["math"] = 82

# Looping through nested data
for student, info in classroom.items():
    avg = sum(info["grades"].values()) / len(info["grades"])
    print(f"{student}: Average = {avg:.1f}")
```

JSON & APIs

When you fetch data from a web API, the response is almost always JSON — which Python automatically converts to nested dictionaries. Understanding nested dicts is essential for working with any modern web service, from weather APIs to social media platforms.

Configuration Files

Application settings often use nested dictionaries to organize options by category: `config["database"]["host"]`, `config["logging"]["level"]`. This pattern keeps settings organized and easy to access programmatically.

Data Modeling

Before you learn about classes and objects, nested dictionaries serve as a lightweight way to model entities with multiple attributes. They're perfect for prototyping and scripting before committing to more formal data structures.

HANDS-ON PRACTICE

Exercises — Test Your Knowledge!

Put everything you've learned into practice with these exercises. They progress from basic operations to more challenging tasks. Try solving each one on your own before looking at the hints. Remember: the best way to learn programming is by writing code!

1

Create & Access

Create a dictionary called `book` with keys: `"title"`, `"author"`, `"year"`, and `"pages"`. Fill it with info about your favorite book. Then print each value using both bracket notation and `.get()`.

2

Modify & Expand

Starting with `menu = {"burger": 8.99, "fries": 3.49}`, add `"soda"` at 1.99, change `"burger"` to 9.49, remove `"fries"`, and print the final menu with a loop.

3

Word Counter

Write a program that takes the string `"apple banana apple cherry banana apple"` and creates a dictionary counting how many times each word appears. Expected: `{"apple": 3, "banana": 2, "cherry": 1}`.

4

Nested Challenge

Create a nested dictionary for 3 employees with `"name"`, `"department"`, and `"salary"`. Write a loop that finds and prints the employee with the highest salary.

🌟 Bonus Challenge: Grade Calculator

```
# Build this program step by step:
# 1. Create a dict of student names
#    mapped to a list of test scores
# 2. Calculate each student's average
# 3. Assign letter grades:
#    A (90+), B (80-89), C (70-79),
#    D (60-69), F (below 60)
# 4. Store results in a new dictionary
# 5. Print a formatted report card

students = {
    "Alice": [95, 88, 92, 85],
    "Bob": [72, 68, 75, 80],
    "Charlie": [55, 60, 58, 62],
    "Diana": [98, 95, 100, 97]
}

# Your code here!
```

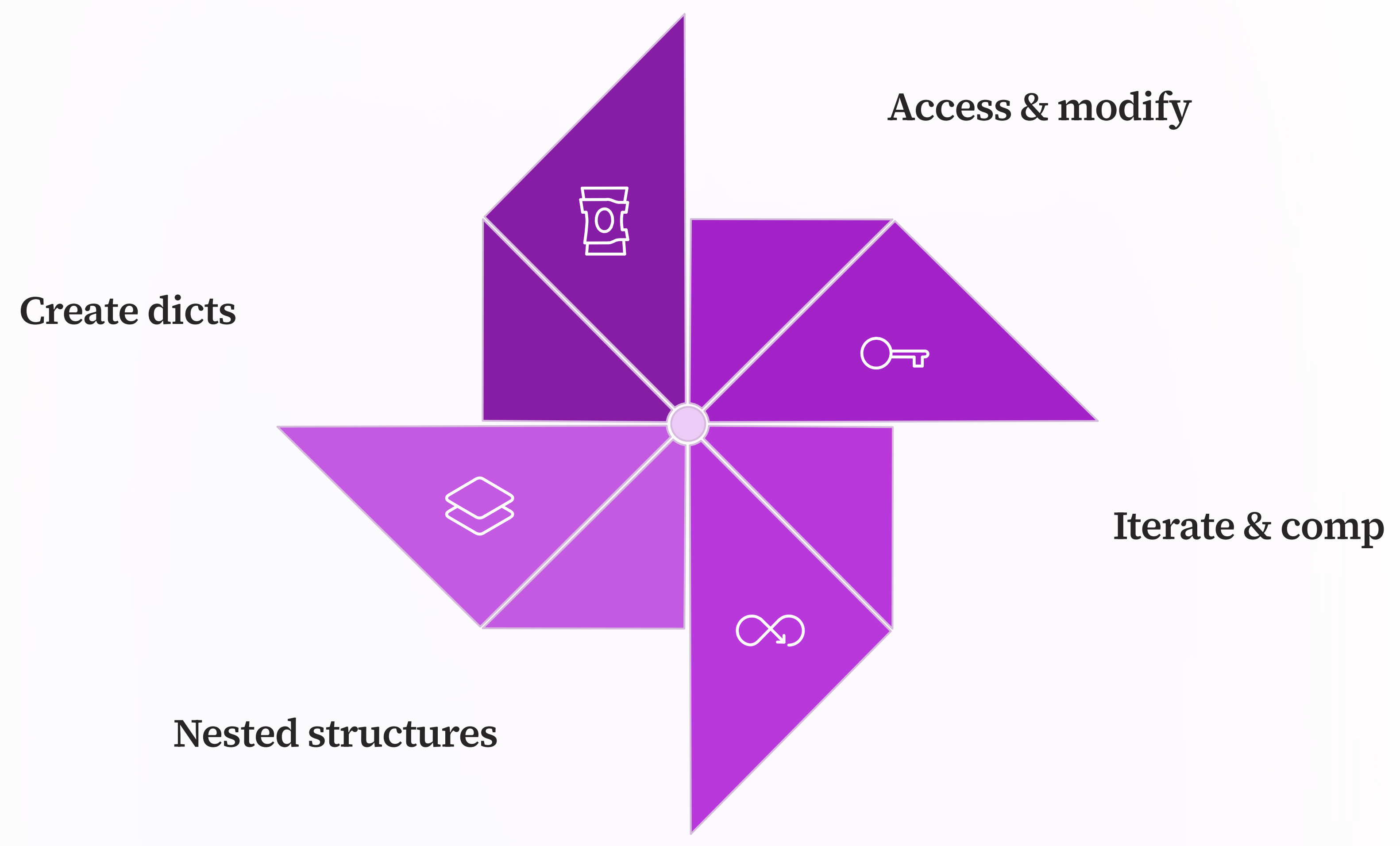
Expected Output

```
===== REPORT CARD =====
Alice:   Avg 90.0 → A
Bob:     Avg 73.8 → C
Charlie: Avg 58.8 → F
Diana:   Avg 97.5 → A
=======================
```

💡 **Hint:** Use `sum(scores) / len(scores)` to calculate averages, and a series of `if/elif` statements for letter grade assignment. Loop through `.items()` to process each student.

Summary & Next Steps

Congratulations! You've covered the essential foundations of Python dictionaries. Let's recap what you've learned and explore where to go from here.



This progression represents your journey from basic dictionary creation to handling real-world data patterns.

Key-Value Pairs

Dictionaries store data as meaningful key-value associations, offering O(1) lookup speed.

Rich Methods

Built-in methods like `.get()`, `.items()`, `.pop()`, and `.update()` make data manipulation effortless.

Comprehensions

Dictionary comprehensions let you create and transform dictionaries in a single, elegant line of code.

Nesting

Nested dictionaries model complex, hierarchical data — the same structure used by JSON and web APIs.

Where to Go Next

01

Sets & Tuples

Explore Python's other built-in data structures. Sets offer unique-element collections, and tuples provide immutable sequences — both complement dictionaries beautifully.

02

File I/O with JSON

Learn to read and write JSON files using Python's `json` module. Since JSON maps directly to dictionaries, your knowledge transfers instantly.

03

Object-Oriented Programming

Graduate from dictionaries to classes and objects. You'll see how the concepts of key-value mapping evolve into attributes and methods on custom objects.

> "The dictionary is one of Python's greatest strengths. Master it, and you've mastered a tool you'll use in virtually every Python program you write." — *Practical advice for every beginner*