2025 Edition

Core Python Unleashed

Where Code Meets Creativity: Code. Create. Innovate.

Kushal Prasad Joshi

kushalprasadjoshi@gmail.com

PREFACE

WELCOME

Welcome to this handbook, a meticulously crafted resource created independently to assist learners, practitioners, and enthusiasts in the field of Programming and Software Development. This guide reflects my passion for Python and represents my personal journey toward mastering the language. I invite you to embark on this adventure of learning, experimentation, and growth.

INSPIRATION

Welcome to this handbook, a meticulously crafted resource created independently to assist learners, practitioners, and enthusiasts in the field of Programming and Software Development. This guide reflects my passion for Python and represents my personal journey toward mastering the language. I invite you to embark on this adventure of learning, experimentation, and growth.

PURPOSE

The goal of this guide is not only to introduce you to Python's core concepts but also to bridge the gap between theoretical knowledge and practical application. By providing in-depth explanations, step-by-step examples, and hands-on exercises, this handbook aims to empower you to:

- Understand the fundamental principles of Python programming.
- Apply Python to solve real-world problems.
- Develop a mindset geared toward continuous learning and innovation.
- Build up a career path in different fields of computer science.

INTERACTIVE LEARNING & GITHUB REPOSITORY

This handbook is an integral part of a GitHub repository, designed to make your learning experience as interactive as possible. All code examples, exercises, and additional resources are available within the repo. For an enhanced, hands-on experience, I encourage you to clone the repository and follow along as you read through the handbook. Experimenting with live code will not only reinforce the concepts presented but also enable you to see Python in action.

Link: https://github.com/KushalPrasadJoshi/core-python-guide

🌟 WHY LEARN PYTHON?

Python is the **#1 programming language** for beginners and professionals alike because of its:

✅ **Simple syntax** (easy to read and write).

✅ **Versatility** (used in web dev, data science, AI, automation, and more).

✅ **Huge community support** (libraries, frameworks, and tutorials).

✅ **High demand** in the job market.

🌱 YOUR LEARNING JOURNEY

Learning a programming language is an evolving journey, filled with challenges and rewarding breakthroughs. This handbook is designed to be your companion on that journey, encouraging you to:

- Experiment with the code examples provided.
- Tackle exercises that challenge your problem-solving skills.
- Reflect on your progress and identify areas for further exploration.

Remember, every programmer starts with a single line of code, and every challenge is an opportunity to learn something new. Embrace the process, stay curious, and enjoy the journey of transforming challenges into achievements.

🙏 THANK YOU & HAPPY LEARNING

Thank you for choosing this resource to accompany you in your quest for Python mastery. I am confident that the insights, practical examples, and structured content presented here will help you build a strong foundation in programming. May this handbook serve as a valuable guide that inspires creativity, fosters innovation, and supports your ongoing learning.

Happy Learning!

Kushal Prasad Joshi

kushalprasadjoshi@gmail.com

Author

TABLE OF CONTENTS

INTRODUCTION TO PYTHON

Welcome to the **Introduction to Python**! This unit covers Python's history, a quick installation guide, and writing your first program. Let's dive in!

HISTORY OF PYTHON

Python was conceived in the late 1980s, and its implementation began in December 1989 by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands. It was designed to be an easy-to-read language that emphasizes readability and simplicity, making it accessible to beginners and powerful for experts.

KEY MILESTONES

1. **1991**: Created by **Guido van Rossum** to emphasize code readability.
2. **2000**: Python 2.0 released (legacy version, now deprecated).
3. **2008**: Python 3.0 released (backward-incompatible, current standard).
4. **2020**: Python 3.9 introduced faster updates and new syntax features.

TIMELINE

- **1991** → Python 1.0
- **2000** → Python 2.0
- **2008** → Python 3.0
- **2023** → Python 3.11 (Latest)

💡 **Fun Fact**: Python is named after the British comedy series *Monty Python*!

GETTING STARTED TO PROGRAM

Before starting to write a program, we need a Python interpreter and a code editor. The Python interpreter is essential for executing Python code, while a code editor provides a user-friendly environment to write and manage your scripts. Popular code editors include Visual Studio Code, PyCharm, and Sublime Text.

QUICK INSTALLATION GUIDE

1. **Download Python:** Visit python.org/downloads.
2. **Install:**
   - Check **"Add Python to Path"** during setup (critical for terminal access).
3. **Verify:**

```
python –version
# Output: Python 3.x.x
```

YOUR FIRST PYTHON PROGRAM

METHOD 1: INTERACTIVE SHELL

```
>>> print("Hello, Python!")
Hello, Python!
```

METHOD 2: SCRIPT FILE

1. **Create "hello.py":**

```
print("Hello, Python!")
```

2. **Run it:**

```
python hello.py
```

PYTHON VS. OTHER LANGUAGES

Python is known for its cleaner syntax and its extensive libraries. Python's standard library is vast and includes modules for various tasks such as web development, data analysis, machine learning, and more. This makes Python a versatile language suitable for many different applications.

EXAMPLE: "HELLO WORLD" COMPARISON

Language	Code	Lines
Python	print("Hello World")	1

Java	public class HelloWorld {
	public static void main(String[] args) {
	System.out.println("Hello World");
	}
	}
		5

C++	#include <iostream>
	int main() {
	std::cout << "Hello World");
	return 0;
	}
		5

Python Wins: Cleaner syntax, fewer lines, no boilerplate!

BASIC INPUT/OUTPUT

Input and output are fundamental operations in any programming language. They allow a program to interact with the user or other systems. In Python, these operations are straightforward and easy to use.

OUTPUT WITH PRINT()

```
print("Welcome to Python!")  # Output: Welcome to Python!
```

INPUT WITH INPUT()

```
name = input("What's your name? ")
print(f"Hello, {name}! 👋")
```

Example Output:

```
What's your name? Alice
Hello, Alice! 👋
```

FORMATTED STRINGS

Formatted strings in Python allow you to embed expressions inside string literals, using curly braces {}. This makes it easy to create strings that include variable values or expressions.

USING F-STRINGS

Introduced in Python 3.6, f-strings (formatted string literals) are prefixed with an f and use curly braces to embed expressions.

```
name = "Alice"
age = 30
print(f"Hello, {name}! You are {age} years old.")
```

Output:

```
Hello, Alice! You are 30 years old.
```

USING STR.FORMAT()

The str.format() method is another way to format strings. It uses curly braces as placeholders which are replaced by the arguments passed to the method.

```
name = "Bob"
age = 25
print("Hello, {}! You are {} years old.".format(name, age))
```

Output:

```
Hello, Bob! You are 25 years old.
```

USING % OPERATOR

The % operator is an older method for string formatting. It uses % followed by a format specifier.

```
name = "Charlie"
age = 22
print("Hello, %s! You are %d years old." % (name, age))
```

Output:

```
Hello, Charlie! You are 22 years old.
```

Formatted strings make it easy to create dynamic and readable strings in Python.

🚨 COMMON PITFALLS

1. INPUT VALUE

```
a = input("Enter first number: ")
b = input("Enter second number: ")
print("The sum is:", a + b)  # Error: This concatenates strings, not adds numbers!
```

✅ **Fix:** Convert types explicitly:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print("The sum is:", a + b)
```

🌍 REAL-WORLD SCENARIO: TEMPERATURE CONVERTER

Let's build a simple Celsius-to-Fahrenheit converter:

```
# Get user input
celsius = float(input("Enter temperature in Celsius: "))

# Convert to Fahrenheit
fahrenheit = (celsius * 9/5) + 32

# Display result
print(f"{celsius}°C = {fahrenheit}°F")
```

Output:

```
Enter temperature in Celsius: 25
25.0°C = 77.0°F
```

✅ TRY IT YOURSELF:

1. **Modify the temperature converter to accept Fahrenheit and convert to Celsius!**

```
# Get user input
celsius = float(input("Enter temperature in Celsius: "))

# Convert to Fahrenheit
fahrenheit = (celsius * 9/5) + 32

# Display result
print(f"{celsius}°C = {fahrenheit}°F")
```

📚 RESOURCES

- **Official Python Documentation**: docs.python.org
- **PEP 8 Style Guide**: Python Code Standards
- **Python Community**: Python.org Community

PYTHON DATA TYPES & VARIABLES

Learn Python's core data types, variables, type conversions, and common pitfalls.

DATA TYPES CHEAT SHEET

Type	Syntax	Mutable?	Example	Use Case
Integers	int	No	age = 25	Counting, calculations
Float	float	No	price = 19.99	Measurements, decimals
String	str	No	name = "Alice"	Text processing
Boolean	bool	No	is_valid = True	Flags, conditions
List	list	Yes	colors = ["red", "blue"]	Dynamic collections
Tuple	tuple	No	coords = (10, 20)	Immutable data groups
Set	set	Yes	unique = {1, 2, 3}	Unique items, membership checks
Dictionary	dict	Yes	user = {"name": "Alice"}	Key-value storage

COMMON PITFALLS

1. TYPE ERRORS

```
num = "5"
total = num + 10  # Error: Can't add str and int!
```

✅ **Fix**: Convert types explicitly:

```
total = int(num) + 10
```

2. MUTABLE DEFAULT ARGUMENTS

```
def add_item(item, items=[]):
    items.append(item)
    return items

# Repeated calls reuse the same list!
print(add_item("apple"))  # ["apple"]
print(add_item("banana"))  # ["apple", "banana"]
```

✅ **Fix**: Use None as default:

```
def add_item(item, items=None):
```

```
    items = items or []
    items.append(item)
    return items
```

3. MODIFYING LISTS DURING ITERATION

```
numbers = [1, 2, 3, 4]
for num in numbers:
    if num % 2 == 0:
        numbers.remove(num)  # Skips elements!
```

✅ **Fix**: Iterate over a copy:

```
for num in numbers.copy():
    if num % 2 == 0:
        numbers.remove(num)
```

🔄 TYPE CONVERSION

EXPLICIT CONVERSION

```
age = int("25")  # String ➔ Integer
price = float("19.99")  # String ➔ Float
text = str(100)  # Integer ➔ String
```

REAL-WORLD EXAMPLE: USER INPUT TO NUMBER

```
user_input = input("Enter your age: ")
age = int(user_input)  # Convert input to integer
print(f"You'll be {age + 10} in 10 years!")
```

💡 TYPE ANNOTATIONS (PYTHON 3.5+)

Type hints improve code readability and enable static type checkers like mypy:

```
def greet(name: str) -> str:
    return f"Hello, {name}"

# Annotating variables
age: int = 25
grades: list[float] = [90.5, 85.0, 77.5]
```

⚠️ **Note**: Python remains dynamically typed annotations are optional and not enforced at runtime.

🌍 REAL-WORLD SCENARIOS

1. SHOPPING CART (LIST)

```
cart = ["apples", "milk", "bread"]
cart.append("eggs")  # Add item
cart.pop(1)  # Remove "milk"
```

2. STUDENT RECORD (DICTIONARY)

```
student = {
    "name": "Alice",
    "courses": ["Math", "Physics"],
    "grades": {"Math": 90, "Physics": 85}
}
print(student["grades"]["Math"])  # 90
```

✅ TRY IT YOURSELF!

1. **Fix the Bug:**

```
# This code raises an error. Fix it!
a = "10"
b = 20
print(a + b)
```

2. **User Profile**:
   Create a dictionary user_profile with keys: name, email, age.

3. **List Conversion**:

   Convert ["10", "20", "30"] to a list of integers.

📚 RESOURCES

- **Python Docs**: Built-in Types
- **PEP 484**: Type Hints

PYTHON OPERATORS

Master Python operators for efficient calculations, comparisons, and logical workflows.

OPERATOR CHEAT SHEET

Category	Operator	Syntax	Example
Arithmetic	+, -, *, /, //, %, **	a + b	5 + 3 → 8
Comparison	==, !=, >, <, >=, <=	a == b	5 == 3 → False
Logical	and, or, not	x and y	True and False → False
Bitwise	&, |, ^, ~, <<, >>	a & b	5 & 3 → 1
Assignment	=, +=, -=, *=	a += 5	a = 10 → 15
Identity	is, is not	a is b	"hi" is "hi" → True
Membership	in, not in	x in list	2 in [1,2,3] → True

COMMON PITFALLS

1. MISUSING IS FOR VALUE COMPARISON

```
a = 256
b = 256
print(a is b)  # True (due to interning)

c = 257
d = 257
print(c is d)  # False (for larger integers)
```

✅ **Fix**: Use == for value checks: print(a == b).

2. OPERATOR PRECEDENCE ERRORS

```
result = 5 + 3 * 2  # 11, not 16!
```

✅ **Fix**: Use parentheses: (5 + 3) * 2 → 16.

3. CHAINED COMPARISONS

```
# Valid in Python!
if 0 < age <= 100:
    print("Valid age")
```

REAL-WORLD SCENARIOS

1. SHOPPING CART TOTAL (ARITHMETIC)

```
price = 49.99
quantity = 3
discount = 0.15
total = (price * quantity) * (1 - discount)
print(f"Total: ${total:.2f}")  # $127.47
```

2. USER AUTHENTICATION (LOGICAL)

```
has_valid_email = True
has_password = False
access = has_valid_email and has_password  # False
```

3. BITMASK PERMISSIONS (BITWISE)

```
READ = 0b100
WRITE = 0b010
EXECUTE = 0b001

user_perms = READ | WRITE
print(bin(user_perms))  # 0b110 (READ + WRITE)
```

✅ TRY IT YOURSELF!

1. **Fix the bug:**

```
# Why does this return False?
a = 10
b = 10.0
print(a is b)
```

2. **Calculate BMI:**

Use an arithmetic operator to compute BMI.

3. **Check Username Validity:**

   Use and/or to validate:

   - Length >= 6
   - Contains letters and numbers.

📚 RESOURCES

- **Python Docs**: Expressions
- **Operator Precedence**: Order of Operations

PYTHON STRINGS

Master Python strings with syntax, methods, real-world use cases, and common pitfalls.

STRING BASICS CHEAT SHEET

Feature	Syntax	Example
Single Quotes	'text'	'Python'
Double Quotes	"text"	"Hello"
Triple Quotes	'''text''' or """text"""	Multi-line strings
Escape Sequences	\n, \t, \", \\	"Line 1\nLine 2"
Raw Strings	r"text"	r"C:\path" (ignores escapes)
Indexing	s[index]	"Python"[0] → "P"
Slicing	s[start:end:step]	"Python"[0:3] → "Pyt"

COMMON PITFALLS

1. STRING IMMUTABILITY

```
s = "hello"
s[0] = "H"  # Error: Strings are immutable!
```

✅ **Fix**: Create a new string.

```
s = "H" + s[1:]  # "Hello"
```

2. ENCODING/DECODING ISSUES

```
data = b"Hello"  # Bytes object
text = "Hello"    # Unicode string
print(data.decode("utf-8") == text)  # True
```

STRING METHODS CHEAT SHEET

Method	Action	Example
s.upper()	Uppercase	"Hi".upper() → "HI"
s.lower()	Lowercase	"Hi".lower() → "hi"
s.strip()	Remove whitespace	" Hi ".strip() → "Hi"
s.split(delim)	Split into list	"a,b,c".split(",") → ["a","b","c"]
s.join(list)	Join list into string	",".join(["a","b"]) → "a,b"

s.replace(old, new)	Replace substring	"Hello".replace("e", "a") → "Hallo"
s.find(sub)	Find substring index	"Python".find("th") → 2

🌍 REAL-WORLD SCENARIOS

1. CLEANING USER INPUT

```
user_input = "  ALICE@gmail.com  "
clean_email = user_input.strip().lower()
print(clean_email)  # "alice@gmail.com"
```

2. GENERATING FORMATTED REPORTS (F-STRINGS)

```
name = "Alice"
score = 95
report = f"""
Student: {name.upper()}
Score: {score}/100
Status: {"Pass" if score >= 50 else "Fail"}
"""
print(report)
```

💡 ADVANCED STRING FEATURES

TYPE ANNOTATIONS (PYTHON 3.5+)

```
def reverse_string(s: str) -> str:
    return s[::-1]

print(reverse_string("Python"))  # "nohtyP"
```

RAW STRINGS FOR FILE PATHS

```
path = r"C:\Users\Alice\Documents"  # No escape issues!
```

✅ TRY IT YOURSELF!

1. **Palindrome Check:**
   Write a program to check if a string reads the same backward.
2. **Parse CSV String:**
   Convert "name,age,email\nAlice,25,alice@mail.com" into a list of dictionaries.
3. **Password Validator:**
   Use string methods to check if a password has:
   - ≥ 8 characters

- At least one uppercase and one digit

📚 RESOURCES

- **Python Docs**: String Methods
- **f-strings Guide**: Real Python

PYTHON LISTS AND TUPLES

Learn the differences, use cases, and best practices for Python's two most versatile sequence types.

LISTS VS. TUPLES CHEAT SHEET

Feature	List	Tuple
Mutability	Mutable (can be modified)	Immutable (fixed after creation)
Syntax	Square brackets [ ]	Parentheses ( )
Performance	Slightly slower (dynamic)	Faster (static)
Use Case	Dynamic data (e.g., shopping cart)	Fixed data (e.g., database config)
Methods	.append(), .sort(), .pop(), etc.	.count(), .index() only

COMMON PITFALLS

1. ACCIDENTAL TUPLE MODIFICATION

```
coordinates = (10, 20)
coordinates[0] = 5  # Error: Tuples are immutable!
```

✅ **Fix**: Use a list if you need mutability.

2. SHALLOW COPIES IN LISTS

```
list1 = [[1, 2], [3, 4]]
list2 = list1.copy()
list2[0][0] = 99
print(list1)  # [[99, 2], [3, 4]] (both changed!)
```

✅ **Fix**: Use copy.deepcopy() for nested structures.

CORE OPERATIONS

LIST METHODS CHEAT SHEET

Method	Action	Example
.append(x)	Add item to end	[1,2].append(3) → [1,2,3]

.insert(i, x)	Insert at index i	[1,3].insert(1,2) → [1,2,3]
.remove(x)	Delete first x	[1,2,2].remove(2) → [1,2]
.sort()	Sort in-place	[3,1,2].sort() → [1,2,3]
list[start:end]	Slice sublist	[1,2,3,4][1:3] → [2,3]

TUPLE UNPACKING

```
# Swap variables
a, b = 5, 10
a, b = b, a  # a=10, b=5

# Function returning multiple values
def get_user():
    return ("Alice", 25, "alice@mail.com")
name, age, email = get_user()
```

🌎 REAL-WORLD SCENARIOS

1. SHOPPING CART (LIST)

```
cart = ["apples", "milk"]
cart.append("bread")  # Add item
cart.remove("milk")   # Remove item
print(f"Cart: {', '.join(cart)}")  # Cart: apples, bread
```

2. RGB COLOR CONFIG (TUPLE)

```
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

def set_color(color: tuple):
    print(f"R: {color[0]}, G: {color[1]}, B: {color[2]}")
```

💡 ADVANCED FEATURES

TYPE ANNOTATIONS (PYTHON 3.9+)

```
from typing import List, Tuple

def process_data(
    items: List[str],
    metadata: Tuple[str, int]
```

```
) -> Tuple[List[str], int]:
    return (items, len(items))
```

LIST COMPREHENSIONS

```
# Generate squares of even numbers
squares = [x**2 for x in range(10) if x % 2 == 0]
# [0, 4, 16, 36, 64]
```

✅ TRY IT YOURSELF!

1. **Fix the Bug:**

```
# Why does this fail? Fix it!
config = ("localhost", 8080)
config[1] = 8000
```

2. **Merge Lists:**
   Combine [1, 2, 3] and [4, 5, 6] into [1,2,3,4,5,6] without using +.
3. **Tuple to Dictionary:**
   Convert (("name", "Alice"), ("age", 25)) into {"name": "Alice", "age": 25}.

📚 RESOURCES

- **Python Docs**: Lists | Tuples
- **Efficiency Guide**: When to Use Lists and Tuples

PYTHON DICTIONARIES AND SETS

Learn to leverage dictionaries for structured data and sets for unique elements with practical examples.

DICTIONARIES VS. SETS CHEAT SHEET

Feature	Dictionary (dict)	Set (set)
Purpose	Key-value pairs (e.g., user profiles)	Unique elements (e.g., tags)
Syntax	{key: value}	{item1, item2}
Mutability	Keys: Immutable Values: Mutable	Mutable (except frozenset)
Lookup Speed	O(1) for keys (ultra-fast)	O(1) for membership checks
Order	Insertion order preserved (Python 3.7+)	Unordered

COMMON PITFALLS

1. KEY ERRORS IN DICTIONARIES

```
user = {"name": "Alice"}
print(user["age"])  # KeyError: 'age' not found!
```

✅ **Fix**: Use .get() with a default:

```
age = user.get("age", 25)  # Returns 25 if "age" missing
```

2. MUTABLE KEYS IN DICTIONARIES

```
invalid_dict = {["a", "b"]: "value"}  # Error: Lists can't be keys!
```

✅ **Fix**: Use tuples or strings as keys.

3. SET DUPLICATE ENTRIES

```
nums = {1, 2, 2, 3}  # Duplicates auto-removed: {1, 2, 3}
```

CORE OPERATIONS

DICTIONARY METHODS

Method	Action	Example
.keys()	Get all keys	user.keys() → ["name", "age"]

.values()	Get all values	user.values() → ["Alice", 25]
.items()	Get key-value pairs	user.items() → [("name", "Alice"), ...]
.setdefault()	Safe key access	user.setdefault("city", "Paris")

SET OPERATIONS

Operation	Syntax	Example
Union	set1 | set2	{1,2} | {2,3} → {1,2,3}
Intersection	set1 & set2	{1,2} & {2,3} → {2}
Difference	set1 - set2	{1,2} - {2,3} → {1}

🌎 REAL-WORLD SCENARIOS

1. USER PROFILE (DICTIONARY)

```
user = {
    "id": 101,
    "name": "Alice",
    "roles": {"admin", "editor"}
}
# Add new key
user["email"] = "alice@mail.com"
```

2. DATA DEDUPLICATION (SET)

```
duplicates = [1, 2, 2, 3, 3, 3]
unique = set(duplicates)  # {1, 2, 3}
```

3. JSON SERIALIZATION (DICTIONARY)

```
import json
config = {"host": "localhost", "port": 8080}
json_data = json.dumps(config)  # '{"host": "localhost", "port": 8080}'
```

💡 ADVANCED FEATURES

FROZEN SETS (IMMUTABLE)

```
permissions = frozenset(["read", "write"])
# permissions.add("execute")  # Error: Frozen sets are immutable!
```

TYPE ANNOTATIONS (PYTHON 3.9+)

```
from typing import Dict, Set
```

```
def process_data(
    users: Dict[int, str],
    tags: Set[str]
) -> Dict[int, Set[str]]:
    return {1: {"admin"}, 2: {"editor"}}
```

PERFORMANCE TIP: DICTIONARY LOOKUPS VS. LISTS

```
# Faster with dictionaries!
users_dict = {101: "Alice", 102: "Bob"}
users_list = [101, "Alice", 102, "Bob"]

# Dict lookup: O(1)
print(users_dict[101])  # "Alice"

# List "lookup": O(n)
index = users_list.index(101)
print(users_list[index + 1])  # "Alice"
```

✅ TRY IT YOURSELF!

1. **Merge Dictionaries:**
   Combine {"a": 1} and {"b": 2} into {"a": 1, "b": 2}.
2. **Find Common Elements:**
   Use sets to find common friends between alice_friends = {"Bob", "Charlie"} and bob_friends = {"Charlie", "Diana"}.
3. **Convert Nested Lists to Dictionaries:**
   Turn [["name", "Alice"], ["age", 25]] into {"name": "Alice", "age": 25}.

📚 RESOURCES

- **Python Docs**: Dictionaries | Sets
- **Advanced Structures**: collections Module

CONDITIONAL STATEMENTS

Learn to control program flow with if, elif, else, and modern patterns like match-case.

CONDITIONAL CHEAT SHEET

Statement	Syntax	Use Case
if	if condition:	Single condition check
if-else	if X: ... else: ...	Binary decisions
if-elif-else	if X: ... elif Y: ... else: ...	Multi-condition checks
Ternary	x = a if cond else b	One-liner assignments
match-case (Python 3.10+)	match value: case X:	Structural pattern matching

COMMON PITFALLS & FIXES

1. ACCIDENTAL ASSIGNMENT IN CONDITIONS

```
if x = 5:  # SyntaxError (use ==)
```

2. TRUTHY/FALSY CONFUSION

```
name = ""
if name:  # Falsy (empty string)
    print("Hello!")
```

✅ **Fix**: Explicit checks:

```
if name is not None:
    ...
```

3. NESTED IF HELL

```
if a:
    if b:
        if c:
            ...  # Hard to read!
```

✅ **Fix**: Use and/elif or extract logic into functions.

CORE CONCEPTS

TRUTHY/FALSY VALUES

Falsy	Truthy
0, "", None, [], {}, False	All non-Falsy values

```
user_input = input("Enter text: ")
if user_input:  # True if not empty
    print("Valid input!")
```

SHORT-CIRCUIT EVALUATION

```
# Stops at first False in `and`, first True in `or`
if user.is_admin or (user.age >= 18 and user.has_id):
    grant_access()
```

🌍 REAL-WORLD SCENARIOS

1. GRADING SYSTEM

```
score = 85
if score >= 90:
    grade = "A"
elif score >= 75:
    grade = "B"
elif score >= 50:
    grade = "C"
else:
    grade = "F"
print(f"Grade: {grade}")
```

2. LOGIN VALIDATION

```
username = input("Username: ")
password = input("Password: ")

if username == "admin" and password == "secret":
    print("Login successful!")
else:
    print("Invalid credentials!")
```

3. DISCOUNT ELIGIBILITY

```
is_member = True
purchase_amount = 150

if is_member and purchase_amount > 100:
    discount = 0.15
elif purchase_amount > 200:
```

```
    discount = 0.10
else:
    discount = 0
```

💡 ADVANCED FEATURES

TERNARY OPERATOR

```
age = 20
status = "Adult" if age >= 18 else "Minor"
```

STRUCTURAL PATTERN MATCHING (PYTHON 3.10+)

```
http_status = 404
match http_status:
    case 200 | 201:
        print("Success!")
    case 404:
        print("Not Found")
    case _:
        print("Unknown error")
```

PERFORMANCE TIP: PREFER ELIF OVER NESTED IF

```
# Faster and cleaner
if x > 0:
    ...
elif x == 0:
    ...
else:
    ...
```

✅ TRY IT YOURSELF!

1. **Leap Year Checker:**
   Write a condition to check if a year is a leap year.

```
year = 2024
is_leap = (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
```

2. **Refactor Nested if:**
   Simplify:

```
if user.is_active:
    if user.has_subscription:
        print("Access granted")
```

3. **Type-Checking:**
   Use isinstance() to validate input types.

```
def add(a, b):
    if isinstance(a, (int, float)) and isinstance(b, (int, float)):
        return a + b
    raise TypeError("Numbers only!")
```

   You can skip this if you find it unfamiliar.

📚 RESOURCES

- **Python Docs**: Control Flow
- **PEP 636**: Structural Pattern Matching

PYTHON LOOPS

Master for and while loops to iterate efficiently, with real-world examples and performance tips.

LOOP CHEAT SHEET

Loop Type	Syntax	Use Case
for	for item in iterable:	Iterate over sequences (lists, strings, etc.)
while	while condition:	Repeat until condition becomes False
Nested	Loop inside another loop	Multi-dimensional data (e.g., matrices)
Comprehensions	[x*2 for x in list]	Concise list/dict/set creation

COMMON PITFALLS & FIXES

1. INFINITE WHILE LOOPS

```
while True:  # Runs forever!
    print("Help!")
```

✅ **Fix**: Add an exit condition:

```
count = 0
while count < 5:
    count += 1
```

2. MODIFYING LISTS DURING ITERATION

```
names = ["Alice", "Bob", "Charlie"]
for name in names:
    if "b" in name.lower():
        names.remove(name)  # Skips elements!
```

✅ **Fix**: Iterate over a copy:

```
for name in names.copy():
    ...
```

3. OFF-BY-ONE ERRORS

```
for i in range(5):
    print(i)  # 0-4 (not 1-5!)
```

CORE OPERATIONS

LOOP CONTROL STATEMENTS

Statement	Action
break	Exit loop immediately
continue	Skip to next iteration
pass	Placeholder (do nothing)
else	Execute after loop completes normally

Example:

```
for num in [2, 4, 6]:
    if num % 2 != 0:
        break
else:
    print("All even!")
```

🌍 REAL-WORLD SCENARIOS

1. BATCH PROCESSING (FOR LOOP)

```
sales = [150, 200, 85, 300]
total = 0
for amount in sales:
    if amount >= 100:
        total += amount
print(f"Total high-value sales: ${total}")
```

2. USER INPUT VALIDATION (WHILE LOOP)

```
password = ""
while len(password) < 8 or not any(c.isdigit() for c in password):
    password = input("Enter a strong password: ")
```

3. MATRIX OPERATIONS (NESTED LOOPS)

```
matrix = [[1, 2], [3, 4]]
for row in matrix:
    for num in row:
        print(num * 2, end=" ")  # 2 4 6 8
```

💡 ADVANCED FEATURES

LIST COMPREHENSIONS

```
# Squares of even numbers
```

```
squares = [x**2 for x in range(10) if x % 2 == 0]  # [0, 4, 16, 36, 64]
```

GENERATOR EXPRESSIONS

```
# Memory-efficient iteration
large_data = (x * 2 for x in range(10_000))
```

ENUMERATE FOR INDEX-VALUE PAIRS

```
colors = ["red", "green", "blue"]
for idx, color in enumerate(colors, start=1):
    print(f"{idx}. {color}")
```

✅ TRY IT YOURSELF!

1. **Factorial Calculator:**
   Use a loop to compute 5! = 5*4*3*2*1.
2. **Prime Number Checker:**
   Write a loop to determine if a number is prime.
3. **Password Strength Checker:**
   Validate passwords using:
   - ≥ 8 characters
   - At least 1 uppercase, 1 digit, and 1 special character.

📚 RESOURCES

- **Python Docs**: for | while
- **Efficiency Guide**: Use Loop Like a Pro

FUNCTIONS AND RECURSION

Master function design, recursion, closures, and decorators for clean, reusable, and efficient Python code.

FUNCTION CHEAT SHEET

Feature	Syntax	Use Case
Positional Args	def func(a, b):	Fixed order of inputs
Keyword Args	func(a=1, b=2)	Explicit parameter naming
Default Args	def func(a=0):	Fallback values
***args**	def func(*args):	Variable positional args
****kwargs**	def func(**kwargs):	Variable keyword args
Lambda	lambda x: x*2	Anonymous single-expression functions

COMMON PITFALLS & FIXES

1. MUTABLE DEFAULT ARGUMENTS

```
def append_to(item, items=[]):
    items.append(item)
    return items

print(append_to(1))  # [1]
print(append_to(2))  # [1, 2] (Same list reused!)
```

✅ **Fix**: Use immutable defaults (e.g., None):

```
def append_to(item, items=None):
    items = items or []
    ...
```

2. INFINITE RECURSION

```
def countdown(n):
    countdown(n-1)  # No base case!
```

✅ **Fix**: Define a base case:

```
def countdown(n):
    if n <= 0: return
    countdown(n-1)
```

3. ACCIDENTAL SHADOWING

```
total = 100
def calculate():
    total = 0  # Shadows global "total"
    ...
```

✅ **Fix**: Use global or rename variables.

🔍 CORE CONCEPTS

SCOPE HIERARCHY

Global → Nonlocal (enclosing) → Local

Example:

```
x = "global"
def outer():
    x = "nonlocal"
    def inner():
        nonlocal x  # Refers to outer's x
        x = "local"
    inner()
    print(x)  # "local"
outer()
```

CLOSURES

Functions that retain access to variables from their enclosing scope:

```
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c = counter()
print(c(), c())  # 1, 2
```

🌍 REAL-WORLD SCENARIOS

1. API REQUEST HANDLER (*ARGS, **KWARGS)

```
def api_call(url, *params, **headers):
    print(f"Fetching {url} with params={params} and headers={headers}")

api_call("https://api.com", "data", retries=3, auth="Bearer XYZ")
```

2. RECURSIVE DIRECTORY TRAVERSAL

```
import os

def list_files(path):
    for entry in os.listdir(path):
        full_path = os.path.join(path, entry)
        if os.path.isdir(full_path):
            list_files(full_path)  # Recursion!
        else:
            print(full_path)
```

3. DECORATORS FOR LOGGING

```
def log_time(func):
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time() - start:.2f}s")
        return result
    return wrapper

@log_time
def heavy_computation():
    return sum(i*i for i in range(10**6))
```

💡 ADVANCED FEATURES

TYPE HINTS (PYTHON 3.5+)

```
from typing import Optional

def greet(name: str, age: Optional[int] = None) -> str:
    return f"Hello, {name}" if age is None else f"{name} is {age} years
old"
```

DECORATOR CHAINING

```
def bold(func):
    def wrapper(): return f"<b>{func()}</b>"
```

```
    return wrapper

def italic(func):
    def wrapper(): return f"<i>{func()}</i>"
    return wrapper

@bold
@italic
def hello(): return "Hello"

print(hello())  # <b><i>Hello</i></b>
```

RECURSION LIMITS

```
import sys
sys.setrecursionlimit(3000)  # Default is 1000
```

✅ TRY IT YOURSELF!

1. **Recursive Binary Search:**
   Implement a function to find an item in a sorted list using recursion.
2. **Execution Time Order:**
   Write a decorator to log how long a function takes to run.
3. **Fix the Closure:**
   Debug this code to make the counter work:

```
def counter():
    count = 0
    return lambda: count += 1  # Error!
```

📚 RESOURCES

- **Python Docs**: Defining Functions
- **Recursion Guide**: Real Python

PYTHON ERROR HANDLING

Learn to handle exceptions, create custom errors, and write robust code with best practices.

EXCEPTION HANDLING CHEAT SHEET

Exception	Cause	Example
ValueError	Invalid value for operation	int("abc")
KeyError	Missing dictionary key	d["missing"]
TypeError	Incorrect type used	"a" + 5
FileNotFoundError	File doesn't exist	open("missing.txt")
IndexError	Invalid sequence index	[1,2][3]
ZeroDivisionError	Division by zero	5 / 0

COMMON PITFALLS & FIXES

1. OVERBROAD EXCEPT CLAUSES

```
try:
    risky_operation()
except:  # Catches all exceptions (even KeyboardInterrupt!)
    ...
```

✅ **Fix**: Catch specific exceptions:

```
except (ValueError, TypeError) as e:
    ...
```

2. SWALLOWING EXCEPTIONS

```
try:
    ...
except ValueError:
    pass  # Silent failure!
```

✅ **Fix**: Log or re-raise:

```
except ValueError as e:
    logging.error(e)
    raise
```

3. RESOURCE LEAKS

```
file = open("data.txt")
```

```
# If error occurs here, file is never closed!
file.close()
```

✅ **Fix**: Use context managers (with):

```
with open("data.txt") as file:
    ...
```

🔍 CORE CONCEPTS

```
BaseException
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ValueError
      ├── TypeError
      └── ...
```

EXCEPTION CHAINING (PYTHON 3.3+)

Preserve original traceback when re-raising:

```
try:
    ...
except ConnectionError as e:
    raise RuntimeError("API failed") from e
```

LOGGING EXCEPTIONS

```
import logging

try:
    ...
except ValueError as e:
    logging.exception("Input validation failed: %s", e)
```

🌍 REAL-WORLD SCENARIOS

1. FILE HANDLING WITH CLEANUP

```
try:
    with open("data.csv") as file:
        data = file.read()
except FileNotFoundError:
    print("File not found!")
except UnicodeDecodeError:
    print("Invalid file encoding!")
```

2. API RETRY WITH EXPONENTIAL BACKOFF

```
import time

def retry_api_call(max_retries=3):
    retries = 0
    while retries < max_retries:
        try:
            return make_api_request()
        except ConnectionError:
            retries += 1
            time.sleep(2 ** retries)
    raise RuntimeError("Max retries exceeded")
```

3. INPUT VALIDATION WITH CUSTOM EXCEPTIONS

```
class InvalidInputError(ValueError):
    pass

def validate_age(age):
    if age < 0:
        raise InvalidInputError(f"Invalid age: {age}")
```

💡 ADVANCED FEATURES

CUSTOM EXCEPTION HIERARCHIES

```
class APIError(Exception):
    """Base class for API errors."""

class RateLimitError(APIError):
    """Raised when API rate limit is exceeded."""

class TimeoutError(APIError):
    """Raised when API request times out."""
```

EXCEPTION GROUPS (PYTHON 3.11+)

Handle multiple exceptions simultaneously:

```
try:
    ...
except* (ValueError, TypeError) as eg:
    for e in eg.exceptions:
        print(f"Caught: {e}")
```

ASSERTIONS FOR DEBUGGING

```
def calculate_discount(price):
    assert price >= 0, "Price cannot be negative!"
    ...

# Disable with: python -O script.py
```

✅ TRY IT YOURSELF!

1. **Fix Bare Except:**
   Replace except: with specific exception handling in:

```
try:
    user_input = int(input("Enter a number: "))
except:
    print("Error!")
```

2. **Custom Exception:**
   Create InvalidEmailError for invalid email formats.
3. **Logging Setup:**
   Modify the API retry example to log exceptions to a file.
4. **Context Managers:**
   Use contextlib to safely read/write files.

📚 RESOURCES

- **Python Docs**: Errors & Exceptions
- **Logging Guide**: Real Python

PYTHON FILE HANDLING

Master file I/O operations, work with different file types, and handle large datasets efficiently.

FILE MODES CHEAT SHEET

Mode	Description
r	Read (text, default)
w	Write (creates/truncates)
a	Append
x	Exclusive creation
rb/wb	Read/write (binary)
r+/w+	Read+write

COMMON PITFALLS & FIXES

1. RESOURCE LEAKS

```
file = open("data.txt")  # Risk of not closing!
```

✅ **Fix**: Use context managers:

```
with open("data.txt") as file:
    ...
```

2. ENCODING ISSUES

```
# Fails on non-UTF-8 files
with open("data.txt", "r") as f:
    ...
```

✅ **Fix**: Specify encoding:

```
with open("data.txt", "r", encoding="latin-1") as f:
    ...
```

3. ACCIDENTAL FILE OVERWRITE

```
with open("data.txt", "w") as f:  # Truncates existing file!
    ...
```

✅ **Fix**: Use "a" (append) or check existence first with pathlib.

🔍 CORE OPERATIONS

MODERN PATH HANDLING WITH PATHLIB

```
from pathlib import Path

# Platform-agnostic paths
file_path = Path("data") / "report.csv"
if file_path.exists():
    content = file_path.read_text()
```

READING LARGE FILES IN CHUNKS

```
with open("large.log", "r") as f:
    while chunk := f.read(4096):  # Read 4KB chunks
        process(chunk)
```

BINARY FILE OPERATIONS

```
# Copy image
with open("input.jpg", "rb") as src, open("copy.jpg", "wb") as dest:
    dest.write(src.read())
```

🌎 REAL-WORLD SCENARIOS

1. CSV/JSON HANDLING

```
import csv
import json

# CSV to JSON
with open("data.csv", "r") as csv_file:
    csv_reader = csv.DictReader(csv_file)
    data = list(csv_reader)

with open("data.json", "w") as json_file:
    json.dump(data, json_file, indent=2)
```

2. ROTATING LOG FILES

```
import logging
from logging.handlers import RotatingFileHandler

logger = logging.getLogger(__name__)
handler = RotatingFileHandler("app.log", maxBytes=1e6, backupCount=3)  #
1MB per file
```

```
logger.addHandler(handler)
```

3. ZIP FILE EXTRACTION

```
import zipfile

with zipfile.ZipFile("archive.zip", "r") as zip_ref:
    zip_ref.extractall("extracted/")
```

💡 ADVANCED FEATURES

TEMPORARY FILES

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as tmp:
    tmp.write(b"Temporary data")
    tmp_path = tmp.name  # Access later
```

MEMORY-MAPPED FILES

```
import mmap

with open("large.bin", "rb") as f:
    with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm:
        print(mm[:10])  # First 10 bytes
```

FILE SYSTEM OPERATIONS

```
import shutil

# Copy directory
shutil.copytree("src/", "backup/")

# Delete directory
shutil.rmtree("temp/")
```

✅ TRY IT YOURSELF!

1. **Merge Two Files:**
   Combine file1.txt and file2.txt into merged.txt.
2. **Log Analyzer:**
   Count ERROR entries in a log file.
3. **Directory Size Calculator:**
   Use os.walk to compute total size of a folder.

📚 RESOURCES

- **Python Docs:** File Objects
- **Real Python:** Working With Files

OBJECT-ORIENTED PROGRAMMING (OOP)

Master classes, inheritance, polymorphism, and design patterns to write modular, reusable code.

OOP CHEAT SHEET

Concept	Syntax	Purpose
Class	class MyClass:	Blueprint for objects
Object	obj = MyClass()	Instance of a class
Inheritance	class Child(Parent):	Reuse parent class features
Polymorphism	Override methods	Same method name, different behavior
Encapsulation	self._protected	Hide internal state
Abstraction	ABCs with @abstractmethod	Define interfaces

COMMON PITFALLS & FIXES

1. MUTABLE CLASS ATTRIBUTES

```
class Inventory:
    items = []  # Shared across all instances!

i1 = Inventory()
i2 = Inventory()
i1.items.append("Sword")  # Affects i2.items!
```

✅ **Fix**: Use instance attributes in __init__:

```
def __init__(self):
    self.items = []
```

2. OVERUSING INHERITANCE

```
class ElectricCar(Car, Battery):  # Fragile hierarchy!
    ...
```

✅ **Fix**: Prefer composition:

```
class ElectricCar:
    def __init__(self):
        self.engine = Engine()
        self.battery = Battery()
```

3. MISUNDERSTANDING SELF

```
class User:
    def logout():  # Missing self!
        ...
```

✅ **Fix**: Always include self in instance methods.

🔍 CORE CONCEPTS

OOP VS. PROCEDURAL PROGRAMMING

OOP	Procedural
Data & behavior bundled in objects	Functions operate on separate data
Easier code reuse via inheritance	Reusability via functions/modules
Models real-world entities	Linear, step-by-step execution

DUCK TYPING

```
class Duck:
    def quack(self): print("Quack!")

class Person:
    def quack(self): print("I'm quacking!")

def make_sound(obj):
    obj.quack()

make_sound(Duck())  # Quack!
make_sound(Person())  # I'm quacking!
```

DESIGN PATTERNS

SINGLETON

```
class Singleton:
    _instance = None
    def __new__(cls):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance

s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
```

FACTORY

```
class ButtonFactory:
    @staticmethod
    def create_button(type):
        if type == "windows":
            return WindowsButton()
        elif type == "mac":
            return MacButton()
```

🌍 REAL-WORLD SCENARIOS

1. BANKING SYSTEM (INHERITANCE)

```
class Account:
    def __init__(self, balance):
        self.balance = balance

class SavingsAccount(Account):
    def add_interest(self, rate):
        self.balance *= (1 + rate)

class CheckingAccount(Account):
    def deduct_fee(self, fee):
        self.balance -= fee
```

2. GUI FRAMEWORK (POLYMORPHISM)

```
class Widget:
    def draw(self):
        pass

class Button(Widget):
    def draw(self):
        print("Rendering button...")

class TextField(Widget):
    def draw(self):
        print("Rendering text field...")

# All widgets can be drawn uniformly
for widget in [Button(), TextField()]:
    widget.draw()
```

3. E-COMMERCE PAYMENT GATEWAYS (ABSTRACTION)

```
from abc import ABC, abstractmethod
```

```python
class PaymentGateway(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCardGateway(PaymentGateway):
    def process_payment(self, amount):
        print(f"Processing ${amount} via credit card...")

class PayPalGateway(PaymentGateway):
    def process_payment(self, amount):
        print(f"Processing ${amount} via PayPal...")
```

💡 ADVANCED FEATURES

TYPE ANNOTATIONS (PYTHON 3.9+)

```python
from typing import Generic, TypeVar

T = TypeVar('T')

class Box(Generic[T]):
    def __init__(self, item: T):
        self.item = item

box = Box[int](10)  # Explicit type
```

MAGIC METHODS

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v = Vector(2, 3) + Vector(4, 5)
print(v)  # Vector(6, 8)
```

DATACLASSES (PYTHON 3.7+)

```python
from dataclasses import dataclass
```

```python
@dataclass
class Product:
    id: int
    name: str
    price: float

p = Product(1, "Laptop", 999.99)
```

✅ TRY IT YOURSELF!

1. **Fix Shared Mutable Attribute:**
   Modify the Inventory class to prevent shared items across instances.
2. **Shape Hierarchy:**
   Create Circle and Rectangle classes with area() method.
3. **Singleton Logger:**
   Implement a logging class that allows only one instance.

📚 RESOURCES

- **Python Docs**: Classes
- **Design Patterns**: Refactoring.Guru

PYTHON MODULES AND PACKAGES

Learn to structure projects, create reusable packages, and avoid common import pitfalls.

MODULES VS. PACKAGES CHEAT SHEET

Feature	Modules	Packages
Definition	Single .py file	Directory with __init__.py
Import Syntax	import my_module	from my_package import module
Namespace	Global or local	Hierarchical (e.g., package.subpackage)
Use Case	Small utilities	Large projects, libraries

COMMON PITFALLS & FIXES

1. CIRCULAR IMPORTS

```
# module_a.py
import module_b  # module_b imports module_a → Error!
```

✅ **Fix**: Refactor code or use local imports.

2. SHADOWING STANDARD MODULES

```
# math.py (custom module)
import math  # Imports your math.py, not the standard library!
```

✅ **Fix**: Avoid naming modules after built-in modules.

3. MISSING __INIT__.PY

```
my_package/
    module.py  # Not recognized as a package without __init__.py
```

✅ **Fix**: Add empty __init__.py (or use namespace packages).

CORE CONCEPTS

ABSOLUTE VS. RELATIVE IMPORTS

Type	Syntax	Use Case
Absolute	from package import module	Clarity, avoids ambiguity

Relative	from .subpackage import module	Intra-package imports

NAMESPACE PACKAGES (PEP 420)

Split a package across multiple directories without __init__.py:

```
project/
    dir1/
        my_namespace/
            module1.py
    dir2/
        my_namespace/
            module2.py
```

```
# Both modules are accessible under my_namespace
from my_namespace import module1, module2
```

🌍 REAL-WORLD SCENARIOS

1. CLI TOOL AS A PACKAGE

```
my_cli/
    __init__.py
    cli.py
    utils/
        __init__.py
        helpers.py
```

```
# cli.py
def main():
    print("Running CLI...")

if __name__ == "__main__":
    main()
```

2. WEB API STRUCTURE

```
my_api/
    __init__.py
    routes/
        __init__.py
        users.py
    models/
```

```
        __init__.py
        user.py
    utils/
        __init__.py
        auth.py
```

3. PUBLISHING TO PYPI

1. **Create pyproject.toml:**

```
[build-system]
requires = ["setuptools>=64.0"]
build-backend = "setuptools.build_meta"

[project]
name = "my_package"
version = "0.1.0"
```

2. **Build and upload:**

```
python -m build
twine upload dist/*
```

💡 ADVANCED FEATURES

ENTRY POINTS FOR CLI TOOLS

```
# pyproject.toml
[project.scripts]
my-cli = "my_package.cli:main"
```

Run as: my-cli

TYPE HINTS IN MODULES

```
# math_utils.py
from typing import Union

def add(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
    return a + b
```

SECURITY BEST PRACTICES

- Avoid wildcard imports: from module import *
- Use __all__ to limit exposed names:

```
__all__ = ["public_function"]  # Controls `from module import *`
```

✅ TRY IT YOURSELF!

1. **Fix Circular Imports:**
   Refactor a project where module_a.py imports module_b.py and vice versa.
2. **Create a Namespace Package:**
   Split a package across two directories and import modules from both.
3. **Convert Script to Package:**
   Transform a standalone script into a package with setup.py/pyproject.toml.

📚 RESOURCES

- **Python Docs**: Modules
- **Packaging Guide**: PyPA

FUNCTIONAL PROGRAMMING

Leverage functional programming (FP) concepts like immutability, higher-order functions, and decorators to build robust pipelines.

FP CHEAT SHEET

Tool	Syntax	Purpose
map()	map(func, iterable)	Apply function to all items
filter()	filter(func, iterable)	Filter items by condition
reduce()	reduce(func, iterable)	Aggregate values (from functools)
lambda	lambda x: x*2	Anonymous single-expression functions
@lru_cache	functools.lru_cache	Memoization for expensive calls
partial	functools.partial	Pre-fill function arguments

COMMON PITFALLS & FIXES

1. OVERUSING LAMBDA FOR COMPLEX LOGIC

```
# Hard to read!
sorted_data = sorted(data, key=lambda x: (x[1], -x[0]))
```

✅ **Fix**: Use named functions or operator module:

```
from operator import itemgetter
sorted_data = sorted(data, key=itemgetter(1))
```

2. IGNORING IMMUTABILITY

```
def process(items):
    items.append("new")  # Modifies input!
```

✅ **Fix**: Return new collections:

```
def process(items):
    return [*items, "new"]
```

3. LAZY EVALUATION SURPRISES

```
squares = map(lambda x: x**2, data)
print(list(squares))  # Exhausts the iterator!
```

CORE CONCEPTS

FP VS. IMPERATIVE PERFORMANCE

```
import timeit

# Imperative approach
def sum_squares_imperative(n):
    total = 0
    for i in range(n):
        total += i ** 2
    return total

# FP approach
def sum_squares_fp(n):
    return sum(map(lambda x: x**2, range(n)))

print(timeit.timeit(lambda: sum_squares_imperative(1000)))  # ~0.06s
print(timeit.timeit(lambda: sum_squares_fp(1000)))          # ~0.04s
```

RECURSIVE DECORATORS

```
def retry(max_attempts=3):
    def decorator(func):
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < max_attempts:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    attempts += 1
            raise RuntimeError("Max retries exceeded")
        return wrapper
    return decorator

@retry(max_attempts=2)
def fetch_data(url):
    # Simulate API call
    if "example.com" not in url:
        raise ConnectionError
    return "Data"

fetch_data("https://test.com")  # Retries twice → RuntimeError
```

🌍 REAL-WORLD SCENARIOS

1. DATA TRANSFORMATION PIPELINE

```
from functools import reduce
```

```
data = [1, 2, 3, 4, 5]
pipeline = (
    data
    |> (lambda x: filter(lambda n: n % 2 == 0, x))
    |> (lambda x: map(lambda n: n * 2, x))
    |> (lambda x: reduce(lambda a, b: a + b, x))
)
print(pipeline)  # (2*2 + 4*2) = 12
```

2. PARALLEL PROCESSING WITH MULTIPROCESSING

```
from multiprocessing import Pool

def square(n):
    return n ** 2

with Pool(4) as p:
    results = p.map(square, range(10))  # [0, 1, 4, 9, ..., 81]
```

3. CACHING EXPENSIVE CALLS

```
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(100))  # Computed efficiently!
```

💡 ADVANCED FEATURES

THIRD-PARTY FP LIBRARIES

TOOLZ FOR FUNCTION COMPOSITION

```
from toolz import compose

clean = compose(str.strip, str.lower)
print(clean("  Hello "))  # "hello"
```

FN.PY FOR PATTERN MATCHING

```
from fn import _

# Filter even numbers greater than 10
```

```
filter(_ % 2 == 0 and _ > 10, [5, 8, 12, 15])  # [12]
```

PARTIAL FUNCTION APPLICATION

```
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
print(square(5))  # 25
```

✅ TRY IT YOURSELF!

1. **Rewrite Loop with FP:**
   Convert [x*2 for x in data if x > 0] using map/filter.
2. **Memoized Factorial:**
   Use @lru_cache to optimize a recursive factorial function.
3. **Parallel Word Count:**
   Use multiprocessing.Pool to count words in multiple files.

📚 RESOURCES

- **Functional HOWTO**: Python Docs
- **toolz**: Toolz Documentation

🚀 ADVANCED PYTHON CONCEPTS

Master Python's advanced features to build high-performance, scalable, and secure applications.

🧩 ADVANCED CONCEPTS CHEAT SHEET

Feature	Syntax	Use Case
Generator	yield	Lazy data streams
Regex	re.compile(r"\d+")	Pattern matching
Async/Await	async def, await	Non-blocking I/O
Metaclass	class Meta(type):	Dynamic class creation
Type Hints	def func() -> int:	Static code analysis

🚨 COMMON PITFALLS & FIXES

1. GENERATOR EXHAUSTION

```
data = (x for x in range(3))
print(list(data))  # [0, 1, 2]
print(list(data))  # [] (Already exhausted!)
```

✅ **Fix**: Re-initialize generators or use itertools.tee.

2. CATASTROPHIC BACKTRACKING (REGEX REDOS)

```
# Dangerous pattern: Exponential backtracking
re.match(r"(a+)+b", "aaaaaaaaac")  # Extremely slow!
```

✅ **Fix**: Use atomic groups or simplify patterns:

```
re.match(r"(?>a+)+b", "aaaaaaaaac")  # Fails quickly
```

3. THREAD SAFETY ISSUES

```
from threading import Thread

counter = 0
def increment():
    global counter
    for _ in range(1000):
        counter += 1  # Race condition!

threads = [Thread(target=increment) for _ in range(10)]
```

```
[t.start() for t in threads]
[t.join() for t in threads]
print(counter)  # Likely < 10000!
```

✅ **Fix**: Use threading.Lock() or multiprocessing.

🔍 CORE CONCEPTS

GENERATORS VS. LISTS

Generators	Lists
Lazy evaluation	Eager evaluation
Memory-efficient	Memory-heavy
Single-use	Reusable

CONCURRENCY PERFORMANCE COMPARISON: THREADS VS. PROCESSES VS. ASYNC

Approach	Use Case	Pros	Cons	Library
Threads	I/O-bound tasks	Lightweight	GIL-bound	threading
Processes	CPU-bound tasks	Parallelism	High overhead	multiprocessing
Async	High I/O concurrency	Scalable	Complex setup	asyncio

REGEX SYNTAX CHEAT SHEET

Pattern	Matches	Example
\d	Digit	"A1" → "1"
\w	Word character	"user_1" → "user_1"
^	Start of string	^Hello → "Hello World"
$	End of string	World$ → "Hello World"
(?:...)	Non-capturing group	(?:ab)+ → "abab"

DEBUGGING GENERATORS WITH INSPECT

```
import inspect

gen = (x for x in range(3))
print(inspect.getgeneratorstate(gen))  # GEN_CREATED

next(gen)
print(inspect.getgeneratorstate(gen))  # GEN_SUSPENDED
```

🌍 REAL-WORLD SCENARIOS

1. DATA VALIDATION WITH PYDANTIC

```
from pydantic import BaseModel, ValidationError

class User(BaseModel):
    name: str
    age: int

try:
    user = User(name="Alice", age="twenty")  # ValidationError
except ValidationError as e:
    print(e.json())
```

2. ASYNC WEB SCRAPER WITH AIOHTTP

```
import aiohttp
import asyncio

async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    html = await fetch("https://example.com")
    print(html[:100])

asyncio.run(main())
```

3. ASYNC WEB SERVER WITH ASYNCIO

```
from aiohttp import web

async def handle(request):
    return web.Response(text="Hello, async world!")

app = web.Application()
app.add_routes([web.get("/", handle)])

if __name__ == "__main__":
    web.run_app(app)
```

4. REGEX FOR LOG ANALYSIS

```
import re
```

```
log = "ERROR: [2023-09-20] User 'admin' failed login"
pattern = r"ERROR: \[(\d{4}-\d{2}-\d{2})\] User '(\w+)' failed login"
match = re.search(pattern, log)
if match:
    date, user = match.groups()
    print(f"Alert: {user} failed on {date}")
```

5. DATA STREAMING WITH GENERATORS

```
def read_large_file(file_path):
    with open(file_path) as f:
        for line in f:
            yield line.strip()

# Process 1GB file without loading it into memory
for line in read_large_file("data.csv"):
    process(line)
```

6. METACLASS FOR VALIDATION

```
class ValidatorMeta(type):
    def __new__(cls, name, bases, dct):
        if "validate" not in dct:
            raise TypeError("Missing validate() method")
        return super().__new__(cls, name, bases, dct)

class User(metaclass=ValidatorMeta):
    def validate(self):
        return len(self.name) > 0
```

💡 ADVANCED FEATURES

METACLASSES FOR DYNAMIC CLASSES

```
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Logger(metaclass=SingletonMeta):
    pass

logger1 = Logger()
logger2 = Logger()
```

```
print(logger1 is logger2)  # True
```

MEMORY PROFILING

```
import tracemalloc

tracemalloc.start()
data = [x for x in range(10**6)]
snapshot = tracemalloc.take_snapshot()
for stat in snapshot.statistics("lineno")[:3]:
    print(stat)
```

TYPE HINT GENERICS

```
from typing import Generic, TypeVar

T = TypeVar("T")

class Box(Generic[T]):
    def __init__(self, item: T):
        self.item = item

box = Box[int](10)
```

✅ TRY IT YOURSELF!

1. **Prevent ReDoS:**
   Optimize the regex r"(\d+|\w+)+$" to avoid catastrophic backtracking.
2. **Thread-Safe Counter:**
   Fix the thread example to ensure counter reaches 10000 using locks.
3. **Generator Debugging:**
   Use inspect to track the state of a generator iterating over a large file.
4. **Async File Downloader:**
   Use aiohttp to download multiple URLs concurrently.

📚 RESOURCES

- **Regex Security**: OWASP ReDoS Guide
- **Pydantic Docs**: Data Validation Made Easy
- **Regex Guide**: Regular-Expressions.info
- **Asyncio Docs**: Python Docs

PYTHON APIS AND DATA

Learn to interact with REST/GraphQL APIs, handle JSON/XML, and build robust data pipelines.

APIS & DATA CHEAT SHEET

Tool	Syntax	Purpose
requests	requests.get(url)	Simple HTTP requests
aiohttp	async with session.get(url)	Async HTTP requests
json	json.loads(data)	Parse JSON data
xml.etree	ET.parse("data.xml")	Parse XML data
pydantic	BaseModel	Validate API responses
requests-cache	CachedSession()	Cache API responses

COMMON PITFALLS & FIXES

1. HARDCODING API KEYS

```
API_KEY = "12345"  # Exposed in code!
```

✅ **Fix**: Use environment variables:

```
import os
API_KEY = os.environ["API_KEY"]
```

2. IGNORING RATE LIMITS

```
for _ in range(1000):
    requests.get("https://api.example.com/data")  # Risk of ban!
```

✅ **Fix**: Add delays or use tenacity for backoff:

```
from tenacity import retry, wait_exponential

@retry(wait=wait_exponential(multiplier=1, min=2, max=10))
def safe_request():
    requests.get(...)
```

3. UNVALIDATED API RESPONSES

```
data = response.json()
print(data["missing_key"])  # KeyError!
```

✅ **Fix**: Validate with pydantic:

```
class ResponseModel(BaseModel):
    required_key: str

validated = ResponseModel(**data)
```

🔍 CORE CONCEPTS

HTTP METHODS & STATUS CODES

Method	Use Case	Status	Meaning
GET	Fetch data	200	OK
POST	Create data	201	Created
PUT	Update data	204	No Content
DELETE	Remove data	404	Not Found

JSON VS XML

Feature	JSON	XML
Syntax	{"key": "value"}	<key>value</key>
Readability	High	Verbose
Use Case	APIs	Legacy systems

🌎 REAL-WORLD SCENARIOS

1. WEATHER CLI WITH OPENWEATHERMAP API

```
import requests

def get_weather(city):
    API_KEY = os.environ["OWM_API_KEY"]
    url =
f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid={API_KEY}
"
    response = requests.get(url)
    return response.json()["main"]["temp"]

print(f"Temperature: {get_weather('London')}K")
```

2. ASYNC GITHUB API PAGINATION

```
import aiohttp
import asyncio
```

```
async def fetch_github_repos(username):
    async with aiohttp.ClientSession() as session:
        repos = []
        page = 1
        while True:
            url =
f"https://api.github.com/users/{username}/repos?page={page}"
            async with session.get(url) as response:
                data = await response.json()
                if not data:
                    break
                repos.extend(data)
                page += 1
        return repos

asyncio.run(fetch_github_repos("torvalds"))
```

3. STREAMING DATA WITH WEBSOCKETS

```
import websockets
import asyncio

async def listen_to_stream():
    async with websockets.connect("wss://stream.example.com") as ws:
        while True:
            data = await ws.recv()
            print(f"Received: {data}")

asyncio.run(listen_to_stream())
```

💡 ADVANCED FEATURES

```
from requests_oauthlib import OAuth2Session

client_id = os.environ["CLIENT_ID"]
client_secret = os.environ["CLIENT_SECRET"]

oauth = OAuth2Session(client_id, redirect_uri="https://callback.com")
authorization_url, _ =
oauth.authorization_url("https://auth.example.com")
print(f"Authorize here: {authorization_url}")

# After user grants access:
token = oauth.fetch_token("https://auth.example.com/token",
client_secret=client_secret)
response = oauth.get("https://api.example.com/user")
```

GRAPHQL API CLIENT

```
from gql import gql, Client
from gql.transport.requests import RequestsHTTPTransport

transport = RequestsHTTPTransport(url="https://api.example.com/graphql")
client = Client(transport=transport)

query = gql("""
    query GetUser($id: ID!) {
        user(id: $id) {
            name
            email
        }
    }
""")

result = client.execute(query, variable_values={"id": 1})
```

CACHING API RESPONSES

```
from requests_cache import CachedSession

session = CachedSession("api_cache", expire_after=3600)  # Cache for 1
hour
response = session.get("https://api.example.com/stable-data")
```

✅ TRY IT YOURSELF!

1. **Stock Price CLI:**
   Use the Alpha Vantage API to fetch real-time stock prices.
2. **GraphQL Pagination:**
   Implement cursor-based pagination for a GraphQL query.
3. **Secure Key Storage:**
   Encrypt API keys using cryptography and load them at runtime.

📚 RESOURCES

- **OAuth2 Guide**: Auth0 Docs
- **GraphQL Tutorial**: How to GraphQL

PYTHON VIRTUAL ENVIRONMENTS

Master virtual environments to avoid dependency conflicts and ensure project reproducibility.

VIRTUAL ENVIRONMENT CHEAT SHEET

Tool	Command	Purpose
venv	python -m venv myenv	Built-in environment creation
virtualenv	virtualenv myenv	Legacy tool with more features
pip	pip install package	Install packages
pip freeze	pip freeze > requirements.txt	Export dependencies
pipenv	pipenv install package	Combines pip + virtualenv
poetry	poetry add package	Modern dependency management

COMMON PITFALLS & FIXES

1. GLOBAL PACKAGE POLLUTION

```
pip install pandas  # Installs globally → Conflicts!
```

✅ **Fix**: Always use a virtual environment.

2. FORGOT TO ACTIVATE ENVIRONMENT

```
# Installs to global Python!
pip install requests
```

✅ **Fix**: Activate the environment first:

```
source myenv/bin/activate  # Linux/macOS
myenv\Scripts\activate     # Windows
```

3. OUTDATED REQUIREMENTS FILE

```
pip freeze > requirements.txt  # Misses dev dependencies!
```

✅ **Fix**: Use pipenv or poetry to track all dependencies.

CORE CONCEPTS

WHY USE VIRTUAL ENVIRONMENTS?

- **Isolation**: Avoid conflicts between project dependencies.
- **Reproducibility**: Share exact dependency versions via requirements.txt.
- **Security**: Prevent system-wide package tampering.

DEPENDENCY MANAGEMENT TOOLS

Tool	Pros	Cons
venv	Built-in, simple	Limited features
pipenv	Combines pip + virtualenv	Slower, less popular now
poetry	Modern, dependency resolution	Steeper learning curve
conda	Cross-platform, non-Python packages	Heavyweight

🌎 REAL-WORLD SCENARIOS

1. PROJECT-SPECIFIC ENVIRONMENT

```
python -m venv myproject-env
source myproject-env/bin/activate
pip install -r requirements.txt
```

2. REPRODUCIBLE BUILDS WITH POETRY

```
poetry init          # Creates pyproject.toml
poetry add requests  # Adds to dependencies
poetry install       # Installs with lock file
```

3. DEPENDENCY SECURITY SCANNING

```
pip install safety
safety check -r requirements.txt  # Checks for vulnerabilities
```

💡 ADVANCED FEATURES

VENV WITH CUSTOM PYTHON VERSIONS

```
# Use a specific Python version
python3.9 -m venv myenv
```

INTEGRATION WITH DOCKER

```
FROM python:3.9

# Create and activate venv
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
```

```
COPY requirements.txt .
RUN pip install -r requirements.txt
```

CONDA FOR DATA SCIENCE

```
conda create -n myenv python=3.9 numpy pandas
conda activate myenv
```

✅ TRY IT YOURSELF!

1. **Migrate to poetry:**
   Convert an existing project using requirements.txt to poetry.
2. **Fix Dependency Hell:**
   Resolve version conflicts using poetry add package@^1.2.3.
3. **Dockerize a Python App:**
   Create a Docker image that uses a virtual environment.

📚 RESOURCES

- **Python Docs**: venv
- **Poetry Docs**: Dependency Management

Congratulations!

You've completed the Core Python!

Thank you for reaching the end.

Do share with friends if you find it helpful.

Share your feedback if possible.

Get more information about python here:

https://github.com/KushalPrasadJoshi/core-python-guide

Get more study materials here:

https://github.com/KushalPrasadJoshi/study-resources

You can follow me on Linked In here:

https://www.linkedin.com/in/kushal-prasad-joshi/