Python Programming

Complete Guide

Beginner → Intermediate → Advanced

13 Comprehensive Chapters | 100+ Code Examples | Real-World Projects

Data Types • OOP • File I/O • NumPy • Pandas • Decorators • Generators

Table of Contents

1. Python Basics
2. Data Types & Variables
3. Operators
4. Control Flow
5. Functions
6. Data Structures
7. Strings
8. File Handling
9. Object-Oriented Programming
10. Advanced Python
11. Libraries for Data Science
12. Exception Handling
13. Real-World Projects

Chapter 1

Python Basics

1.1 Introduction to Python

What is Python?

Python is a high-level, interpreted, general-purpose programming language created by **Guido van Rossum** and first released in **1991**. It emphasises code readability and simplicity, allowing developers to write fewer lines of code compared to languages like C++ or Java. Python uses indentation (whitespace) to define code blocks rather than curly braces, which enforces a clean and consistent style.

Key Features of Python

- **Simple & Readable:** English-like syntax makes it easy to learn.
- **Interpreted:** Code is executed line-by-line; no compilation step required.
- **Dynamically Typed:** No need to declare variable types explicitly.
- **Object-Oriented:** Supports classes, inheritance, and encapsulation.
- **Extensive Libraries:** Thousands of built-in and third-party modules.
- **Cross-Platform:** Runs on Windows, macOS, Linux, and more.
- **Open Source & Free:** Large, active community worldwide.

Applications of Python

Domain	Examples
Web Development	Django, Flask, FastAPI
Data Science	Pandas, NumPy, Matplotlib
Machine Learning / AI	TensorFlow, PyTorch, Scikit-learn
Automation & Scripting	Selenium, Paramiko
Cybersecurity	Scapy, Nmap integration
Scientific Computing	SciPy, SymPy
Game Development	Pygame

■ Real-World Use Case

Python powers Instagram's backend (Django), Netflix's recommendation engine, NASA's data analysis pipelines, and Google's internal tools — making it one of the most versatile languages in the industry.

1.2 Installation & Setup

Installing Python

1. Visit **https://www.python.org/downloads/** and download the latest stable version (3.12+).
2. On Windows: run the installer, tick **'Add Python to PATH'**, then click Install Now.
3. On macOS/Linux: Python 3 is often pre-installed. Verify with:

■ Terminal

```
python3 --version
```

■ Output

```
Python 3.12.0
```

Setting Up an IDE

IDE	Best For	Install Command / Link
VS Code	General development	https://code.visualstudio.com
Jupyter Notebook	Data science, exploration	pip install notebook
PyCharm	Large projects, debugging	https://jetbrains.com/pycharm
Google Colab	Cloud, ML experiments	https://colab.research.google.com

1.3 Your First Python Program

Every programmer's journey begins with printing "Hello, World!" — a simple program that confirms your environment is set up correctly.

■ hello_world.py

```
# This is a comment
print("Hello, World!")
print("Welcome to Python Programming!")
```

■ Output

```
Hello, World!
Welcome to Python Programming!
```

Explanation

print() is a built-in function that displays text to the console. Strings (text) are enclosed in single or double quotes. Lines starting with # are comments — they are ignored by Python and used to explain code.

Chapter 2

Data Types & Variables

2.1 Variables

A **variable** is a named storage location in memory that holds a value. In Python, you don't need to declare a type — the interpreter infers it automatically.

Rules for Naming Variables

- Must start with a letter (a–z, A–Z) or underscore (_).
- Cannot start with a digit.
- Can contain letters, digits, and underscores.
- Case-sensitive: **age**, **Age**, and **AGE** are three different variables.
- Cannot use Python reserved keywords (if, for, while, class, etc.).

variables.py

```
# Valid variable assignments
name = "Alice"
age = 25
height = 5.7
is_student = True

# Multiple assignment
x = y = z = 0

# Swap variables
a, b = 10, 20
a, b = b, a
print(a, b) # 20 10
```

Output

```
20 10
```

2.2 Data Types

Type: int

Whole numbers (positive, negative, zero).

int_example.py

```
age = 30
temp = -5
big = 1_000_000
```

Output

```
age: 30, type: <class 'int'>
```

Type: float

Decimal (floating-point) numbers.

float_example.py

```
price = 19.99
pi = 3.14159
```

Output

```
price: 19.99, type: <class 'float'>
```

Type: str

A sequence of characters enclosed in quotes.

str_example.py

```
name = "Python"
greeting = 'Hello'
```

Output

```
name: Python, type: <class 'str'>
```

Type: bool

Logical values: True or False.

bool_example.py

```
is_active = True
is_done = False
print(5 > 3)
```

Output

```
True, type: <class 'bool'>
```

Type: NoneType

Represents the absence of a value.

NoneType_example.py

```
result = None
print(result is None)
```

Output

```
True
```

2.3 Type Casting

Implicit Type Conversion

Python automatically converts smaller types to larger types to avoid data loss.

implicit.py

```
x = 5 # int
y = 2.0 # float
```

```
result = x + y # int + float → float
print(result, type(result))
```

■ Output

```
7.0 <class 'float'>
```

Explicit Type Conversion

You manually convert types using built-in functions: int(), float(), str(), bool().

■ explicit_cast.py

```
num_str = "42"
num_int = int(num_str) # str → int
num_float = float(num_str) # str → float
back_str = str(100) # int → str

print(num_int + 8) # 50
print(bool(0), bool(1)) # False True
```

■ Output

```
50
False True
```

■ Real-World Use Case

Reading user input from a form always returns a string. Explicit casting (int(input())) is essential before performing arithmetic on that data — used in every web form, CLI tool, and data-entry application.

Chapter 3

Operators

3.1 Arithmetic Operators

Arithmetic operators perform mathematical calculations.

Operator	Description	Example	Result
+	Addition	10 + 3	13
-	Subtraction	10 - 3	7
*	Multiplication	10 * 3	30
/	Division (float)	10 / 3	3.333...
//	Floor Division	10 // 3	3
%	Modulus (remainder)	10 % 3	1
**	Exponentiation	2 ** 8	256

arithmetic.py

```
a, b = 17, 5
print(a + b, a - b, a * b)
print(a / b, a // b, a % b)
print(a ** 2)
```

Output

```
22 12 85
3.4 3 2
289
```

3.2 Comparison Operators

Comparison operators return boolean values (True or False).

comparison.py

```
x = 10
print(x == 10) # True
print(x != 5) # True
print(x > 7) # True
print(x < 7) # False
print(x >= 10) # True
print(x <= 9) # False
```

Output

```
True
```

```
True
True
False
True
False
```

3.3 Logical Operators

Logical operators combine boolean expressions.

logical.py

```
age = 25
has_id = True

# and: both must be True
print(age >= 18 and has_id) # True

# or: at least one must be True
print(age < 18 or has_id) # True

# not: inverts the boolean
print(not has_id) # False
```

Output

```
True
True
False
```

3.4 Bitwise Operators

Bitwise operators work on the binary representation of integers.

bitwise.py

```
a = 12 # 1100 in binary
b = 10 # 1010 in binary

print(a & b) # AND → 8 (1000)
print(a | b) # OR → 14 (1110)
print(a ^ b) # XOR → 6 (0110)
print(~a) # NOT → -13
print(a << 1) # Left shift → 24
print(a >> 1) # Right shift → 6
```

Output

```
8
14
6
-13
24
```

```
6
```

■ Real-World Use Case

Bitwise operators are used in networking (IP subnet masking), image processing (pixel manipulation), permissions systems in operating systems, and cryptographic algorithms.

Chapter 4

Control Flow

4.1 if Statement

The if statement executes a block of code only when a condition evaluates to True.

if_stmt.py

```
temperature = 35
if temperature > 30:
print("It's hot outside!")
print("Stay hydrated.")
```

Output

```
It's hot outside!
Stay hydrated.
```

4.2 if-else Statement

if_else.py

```
score = 72

if score >= 60:
print("Passed!")
else:
print("Failed. Please retry.")

# elif chain
if score >= 90:
grade = 'A'
elif score >= 80:
grade = 'B'
elif score >= 70:
grade = 'C'
elif score >= 60:
grade = 'D'
else:
grade = 'F'
print(f"Grade: {grade}")
```

Output

```
Passed!
Grade: C
```

4.3 Nested if Statements

■ nested_if.py

```
username = "admin"
password = "1234"

if username == "admin":
if password == "1234":
print("Login successful!")
else:
print("Wrong password.")
else:
print("Unknown user.")
```

■ Output

```
Login successful!
```

4.4 Loops

for Loop

Iterates over a sequence (list, tuple, string, range, etc.).

■ for_loop.py

```
# Loop over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
print(f"I like {fruit}")

# Loop with range
for i in range(1, 6):
print(f"{i} squared = {i**2}")
```

■ Output

```
I like apple
I like banana
I like cherry
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

while Loop

Runs repeatedly as long as the condition remains True.

■ while_loop.py

```
count = 1
```

```
while count <= 5:
print(f"Count: {count}")
count += 1

# break and continue
for n in range(10):
if n == 3:
continue # skip 3
if n == 7:
break # stop at 7
print(n, end=' ')
```

■ Output

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
0 1 2 4 5 6
```

■ Real-World Use Case

Control flow forms the backbone of every program: from authenticating logins (if-else), paginating API results (for loop), to polling a server until data arrives (while loop). It is the decision-making engine of all software.

Chapter 5

Functions

5.1 Function Basics

A function is a reusable, named block of code. Functions improve readability, eliminate repetition (DRY — Don't Repeat Yourself), and make testing easier.

functions_basics.py

```
def greet():
"""This function prints a greeting."""
print("Hello! Welcome to Python.")

# Call the function
greet()
greet() # Call it multiple times
```

Output

```
Hello! Welcome to Python.
Hello! Welcome to Python.
```

5.2 Arguments & Parameters

arguments.py

```
# Positional arguments
def add(a, b):
return a + b
print(add(3, 4)) # 7

# Default arguments
def power(base, exp=2):
return base ** exp
print(power(3)) # 9
print(power(2, 10)) # 1024

# Keyword arguments
def describe(name, age, city):
print(f"{name}, {age} years old, from {city}")
describe(age=28, city="Paris", name="Marie")

# *args and **kwargs
def total(*nums):
return sum(nums)
print(total(1, 2, 3, 4, 5)) # 15
```

Output

```
7
9
1024
Marie, 28 years old, from Paris
15
```

5.3 Return Values

■ return_values.py

```
def divide(a, b):
if b == 0:
return None, "Division by zero!"
return a / b, "OK"

result, msg = divide(10, 2)
print(result, msg)

result, msg = divide(5, 0)
print(result, msg)
```

■ Output

```
5.0 OK
None Division by zero!
```

5.4 Lambda Functions

A lambda is an anonymous, single-expression function defined with the lambda keyword. Useful for short, throwaway functions.

■ lambda.py

```
# Syntax: lambda arguments: expression
square = lambda x: x ** 2
print(square(7)) # 49

# With sorted()
students = [("Alice", 88), ("Bob", 72), ("Charlie", 95)]
students.sort(key=lambda s: s[1], reverse=True)
print(students)

# With map() and filter()
nums = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, nums))
doubled = list(map(lambda x: x * 2, nums))
print(evens)
print(doubled)
```

■ Output

```
49
```

```
[('Charlie', 95), ('Alice', 88), ('Bob', 72)]
[2, 4, 6]
[2, 4, 6, 8, 10, 12]
```

■ Real-World Use Case

Lambda functions are heavily used in data processing pipelines — e.g., pandas apply(), sorting leaderboards, event callbacks in GUIs, and inline transformations in data science workflows.

Chapter 6

Data Structures

6.1 Lists

A list is an **ordered, mutable** (changeable) collection of items. Lists can hold mixed data types and allow duplicates.

lists.py

```
fruits = ["apple", "banana", "cherry"]

# Indexing and slicing
print(fruits[0]) # apple
print(fruits[-1]) # cherry
print(fruits[0:2]) # ['apple', 'banana']

# Mutability
fruits.append("mango")
fruits.insert(1, "blueberry")
fruits.remove("banana")
print(fruits)

# Useful methods
nums = [4, 2, 9, 1, 7]
nums.sort()
print(nums)
print(len(nums), sum(nums), min(nums), max(nums))
```

Output

```
apple
cherry
['apple', 'banana']
['apple', 'blueberry', 'cherry', 'mango']
[1, 2, 4, 7, 9]
5 23 1 9
```

6.2 Tuples

A tuple is an **ordered, immutable** collection. Use tuples for data that should not change (e.g., RGB values, coordinates).

tuples.py

```
point = (3, 7)
rgb = (255, 128, 0)

print(point[0]) # 3
```

```
print(rgb[-1]) # 0

# Tuple unpacking
x, y = point
print(f"x={x}, y={y}")

# Named tuple
from collections import namedtuple
Person = namedtuple('Person', ['name', 'age'])
p = Person('Alice', 30)
print(p.name, p.age)
```

Output

```
3
0
x=3, y=7
Alice 30
```

6.3 Sets

A set is an **unordered collection of unique elements**. Duplicate values are automatically removed. Sets support mathematical set operations.

sets.py

```
colors = {"red", "green", "blue", "red"} # duplicate removed
print(colors)

primes = {2, 3, 5, 7, 11}
evens = {2, 4, 6, 8, 10}

print(primes & evens) # intersection
print(primes | evens) # union
print(primes - evens) # difference
```

Output

```
{'blue', 'green', 'red'}
{2}
{2, 3, 4, 5, 6, 7, 8, 10, 11}
{3, 5, 7, 11}
```

6.4 Dictionaries

A dictionary stores **key-value pairs**. Keys must be unique and immutable; values can be any type. Dictionaries preserve insertion order (Python 3.7+).

dictionaries.py

```
student = {
"name": "Alice",
"age": 22,
```

```
"grades": [90, 85, 92]
}

print(student["name"]) # Alice
print(student.get("gpa", "N/A")) # N/A (safe access)

student["gpa"] = 3.8
student["age"] = 23

for key, value in student.items():
print(f"{key}: {value}")

print(list(student.keys()))
print(list(student.values()))
```

■ Output

```
Alice
N/A
name: Alice
age: 23
grades: [90, 85, 92]
gpa: 3.8
['name', 'age', 'grades', 'gpa']
['Alice', 23, [90, 85, 92], 3.8]
```

■ Real-World Use Case

Dictionaries are the backbone of JSON data (APIs), database records, configuration files, caching (memoization), and word-count frequency analysis in NLP tasks.

Chapter 7

Strings

7.1 String Methods

Strings in Python are immutable sequences of Unicode characters. Python provides a rich set of built-in methods for string manipulation.

■ string_methods.py

```
s = " Hello, Python World! "

# Case methods
print(s.upper())
print(s.lower())
print(s.title())

# Strip whitespace
print(s.strip())
print(s.lstrip())

# Search & replace
print(s.find("Python")) # index of first match
print(s.replace("Python", "Amazing"))
print("Python" in s) # True

# Split and join
words = s.strip().split(" ")
print(words)
print("-".join(words))

# String checks
print("abc123".isalnum())
print("hello".isalpha())
print(" ".isspace())
```

■ Output

```
HELLO, PYTHON WORLD!
hello, python world!
Hello, Python World!
Hello, Python World!
Hello, Python World!
8
Hello, Amazing World!
True
['Hello,', 'Python', 'World!']
Hello,-Python-World!
```

```
True
True
True
```

7.2 String Formatting

■ string_format.py

```
name = "Bob"
age = 30
pi = 3.14159

# f-strings (recommended, Python 3.6+)
print(f"Name: {name}, Age: {age}")
print(f"Pi to 2 decimal places: {pi:.2f}")
print(f"Large number: {1_000_000:,}")

# .format() method
print("Name: {}, Age: {}".format(name, age))
print("{0} is {1} years old. {0} loves Python.".format(name, age))

# % formatting (older style)
print("%s is %d years old" % (name, age))

# Multiline f-string
info = (
f"Name : {name}\n"
f"Age : {age}\n"
f"Pi : {pi:.4f}"
)
print(info)
```

■ Output

```
Name: Bob, Age: 30
Pi to 2 decimal places: 3.14
Large number: 1,000,000
Name: Bob, Age: 30
Bob is 30 years old. Bob loves Python.
Bob is 30 years old
Name : Bob
Age : 30
Pi : 3.1416
```

■ Real-World Use Case

String formatting is used everywhere: generating email templates, building SQL queries, formatting log messages, creating HTML content dynamically, and building user-facing notifications in apps.

Chapter 8

File Handling

Python makes it easy to read, write, and manage files using the built-in open() function. The **with** statement (context manager) ensures files are properly closed even if an error occurs.

8.1 Reading Files

read_files.py

```
# Read entire file
with open("data.txt", "r") as f:
content = f.read()
print(content)

# Read line by line
with open("data.txt", "r") as f:
for line in f:
print(line.strip())

# Read all lines into a list
with open("data.txt", "r") as f:
lines = f.readlines()
print(lines)

# Read CSV-like data
with open("students.csv", "r") as f:
header = f.readline().strip().split(",")
for line in f:
row = line.strip().split(",")
record = dict(zip(header, row))
print(record)
```

Output

```
Line 1: Hello World
Line 2: Python File I/O
['Line 1: Hello World\n', 'Line 2: Python File I/O\n']
```

8.2 Writing Files

Mode	Description
'r'	Read (default) — error if file doesn't exist
'w'	Write — creates new or overwrites existing
'a'	Append — adds to end of existing file

'x'	Create — error if file already exists
'rb'/'wb'	Read/Write in binary mode

write_files.py

```
# Write to a file
with open("output.txt", "w") as f:
f.write("Hello, File!\n")
f.write("Python is great.\n")

# Append to a file
with open("output.txt", "a") as f:
f.write("Appended line.\n")

# Write multiple lines
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("output.txt", "w") as f:
f.writelines(lines)

# Working with JSON
import json
data = {"name": "Alice", "scores": [90, 85, 92]}
with open("data.json", "w") as f:
json.dump(data, f, indent=2)

with open("data.json", "r") as f:
loaded = json.load(f)
print(loaded["name"])
```

Output

```
Alice
```

Real-World Use Case

File handling is used in log management systems (writing/rotating logs), ETL pipelines (reading CSVs, writing processed data), configuration management (JSON/YAML config files), and report generation.

Chapter 9

Object-Oriented Programming

Object-Oriented Programming (OOP) is a paradigm that organises code around **objects** — entities that bundle data (attributes) and behaviour (methods). The four pillars of OOP are Encapsulation, Abstraction, Inheritance, and Polymorphism.

9.1 Classes & Objects

classes.py

```
class Car:
# Class variable (shared by all instances)
wheels = 4

def __init__(self, brand, model, year):
# Instance variables
self.brand = brand
self.model = model
self.year = year
self._speed = 0

def accelerate(self, amount):
self._speed += amount
print(f"{self.brand} accelerates to {self._speed} km/h")

def __str__(self):
return f"{self.year} {self.brand} {self.model}"

def __repr__(self):
return f"Car('{self.brand}', '{self.model}', {self.year})"

# Create objects
car1 = Car("Toyota", "Supra", 2023)
car2 = Car("BMW", "M3", 2024)

print(car1)
car1.accelerate(60)
print(Car.wheels)
print(car2)
```

Output

```
2023 Toyota Supra
Toyota accelerates to 60 km/h
4
2024 BMW M3
```

9.2 Inheritance

Inheritance allows a class (child) to acquire attributes and methods from another class (parent), promoting code reuse.

■ inheritance.py

```
class Animal:
def __init__(self, name):
self.name = name

def speak(self):
return "Some sound"

def __str__(self):
return f"Animal: {self.name}"

class Dog(Animal):
def speak(self):
return f"{self.name} says: Woof!"

class Cat(Animal):
def speak(self):
return f"{self.name} says: Meow!"

class GoldenRetriever(Dog):
"""Multi-level inheritance"""
def fetch(self):
return f"{self.name} fetches the ball!"

animals = [Dog("Rex"), Cat("Luna"), GoldenRetriever("Buddy")]
for animal in animals:
print(animal.speak())
print(animals[2].fetch())
```

■ Output

```
Rex says: Woof!
Luna says: Meow!
Buddy says: Woof!
Buddy fetches the ball!
```

9.3 Polymorphism

Polymorphism allows the same interface (method name) to behave differently based on the object calling it.

■ polymorphism.py

```
class Shape:
def area(self):
return 0
```

```
class Circle(Shape):
def __init__(self, radius):
self.radius = radius
def area(self):
return 3.14159 * self.radius ** 2

class Rectangle(Shape):
def __init__(self, w, h):
self.w, self.h = w, h
def area(self):
return self.w * self.h

class Triangle(Shape):
def __init__(self, base, height):
self.base, self.height = base, height
def area(self):
return 0.5 * self.base * self.height

shapes = [Circle(5), Rectangle(4, 6), Triangle(3, 8)]
for shape in shapes:
print(f"{type(shape).__name__} area: {shape.area():.2f}")
```

■ Output

```
Circle area: 78.54
Rectangle area: 24.00
Triangle area: 12.00
```

9.4 Encapsulation

Encapsulation restricts direct access to internal data. In Python, use: **_name** for protected and **__name** for private attributes.

■ encapsulation.py

```
class BankAccount:
def __init__(self, owner, balance):
self.owner = owner
self.__balance = balance # private

def deposit(self, amount):
if amount > 0:
self.__balance += amount

def withdraw(self, amount):
if 0 < amount <= self.__balance:
self.__balance -= amount
else:
print("Insufficient funds!")

@property
```

```
    def balance(self):
        return self.__balance

acc = BankAccount("Alice", 1000)
acc.deposit(500)
acc.withdraw(200)
print(f"Balance: ${acc.balance}")
acc.withdraw(5000) # Insufficient funds!
```

■ Output

```
Balance: $1300
Insufficient funds!
```

■ Real-World Use Case

OOP is the foundation of every major framework: Django (models), Flask (views), PyGame (sprites), and desktop apps (tkinter widgets). The BankAccount pattern is used in payment gateways and fintech APIs.

Chapter 10

Advanced Python

10.1 List Comprehensions

List comprehensions provide a concise, readable way to create lists in a single line. Syntax: **[expression for item in iterable if condition]**

■ comprehensions.py

```
# Basic: squares of 0-9
squares = [x**2 for x in range(10)]
print(squares)

# With filter: even squares
even_sq = [x**2 for x in range(10) if x % 2 == 0]
print(even_sq)

# Nested: multiplication table
table = [[i*j for j in range(1, 4)] for i in range(1, 4)]
for row in table:
print(row)

# Dict comprehension
word = "hello world"
freq = {ch: word.count(ch) for ch in set(word) if ch != ' '}
print(freq)

# Set comprehension
unique_lens = {len(w) for w in ["cat", "dog", "elephant", "rat"]}
print(unique_lens)
```

■ Output

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 4, 16, 36, 64]
[1, 2, 3]
[2, 4, 6]
[3, 6, 9]
{'h': 1, 'e': 1, 'l': 3, 'o': 2, 'w': 1, 'r': 1, 'd': 1}
{3, 8}
```

10.2 Generators

Generators produce items **lazily** (one at a time) instead of building an entire list in memory. Use **yield** instead of return. Essential for large data streams.

■ generators.py

```
# Generator function
def fibonacci(n):
a, b = 0, 1
for _ in range(n):
yield a
a, b = b, a + b

for num in fibonacci(8):
print(num, end=' ')
print()

# Generator expression (memory efficient)
gen = (x**2 for x in range(1_000_000))
print(next(gen)) # 0
print(next(gen)) # 1
print(next(gen)) # 4

# Infinite generator
def counter(start=0):
n = start
while True:
yield n
n += 1

c = counter(10)
print([next(c) for _ in range(5)])
```

■ Output

```
0 1 1 2 3 5 8 13
0
1
4
[10, 11, 12, 13, 14]
```

10.3 Decorators

A decorator is a higher-order function that wraps another function to extend its behaviour without modifying its source code. Used extensively in frameworks like Flask and Django.

■ decorators.py

```
import time
import functools

# Timing decorator
def timer(func):
@functools.wraps(func)
def wrapper(*args, **kwargs):
start = time.time()
```

```
    result = func(*args, **kwargs)
    end = time.time()
    print(f"{func.__name__} took {end-start:.4f}s")
    return result
    return wrapper

@timer
def slow_sum(n):
    return sum(range(n))

print(slow_sum(1_000_000))

# Login-required decorator
def login_required(func):
@functools.wraps(func)
def wrapper(user, *args, **kwargs):
    if not user.get('logged_in'):
        return "Access denied. Please log in."
    return func(user, *args, **kwargs)
return wrapper

@login_required
def dashboard(user):
    return f"Welcome, {user['name']}!"

print(dashboard({'name': 'Alice', 'logged_in': True}))
print(dashboard({'name': 'Bob', 'logged_in': False}))
```

■ Output

```
slow_sum took 0.0312s
499999500000
Welcome, Alice!
Access denied. Please log in.
```

10.4 Modules & Packages

■ modules.py

```
# Built-in modules
import math
import random
import os
import datetime

print(math.sqrt(144)) # 12.0
print(math.factorial(5)) # 120
print(random.randint(1, 100)) # random integer
print(os.getcwd()) # current directory
print(datetime.date.today()) # today's date
```

```
# Selective import
from math import pi, e, log
print(f"pi={pi:.4f}, e={e:.4f}")

# Creating your own module (mathtools.py)
# def add(a, b): return a + b
# def multiply(a, b): return a * b

# import mathtools
# print(mathtools.add(3, 4))

# Package structure:
# mypackage/
# __init__.py
# utils.py
# models.py
```

■ Output

```
12.0
120
73
/home/user/project
2024-01-15
pi=3.1416, e=2.7183
```

■ Real-World Use Case

Decorators power Flask's @app.route(), Django's @login_required, and @property. Generators process gigabyte log files without loading them into RAM. Modules and packages form the architecture of every Python project.

Chapter 11

Libraries for Data Science

11.1 NumPy

NumPy (Numerical Python) is the foundation of scientific computing in Python. It provides fast, multidimensional array operations powered by C under the hood. NumPy arrays are up to 50x faster than Python lists for numerical work.

numpy_intro.py

```
import numpy as np

# Creating arrays
arr = np.array([1, 2, 3, 4, 5])
matrix = np.array([[1,2,3],[4,5,6],[7,8,9]])
zeros = np.zeros((3, 3))
ones = np.ones((2, 4))
range_arr = np.arange(0, 20, 2)
linspace = np.linspace(0, 1, 5)

print(arr * 2) # element-wise
print(arr.mean(), arr.std())
print(matrix.shape) # (3, 3)
print(matrix.T) # transpose
print(matrix[1, 2]) # element at row 1, col 2
print(matrix[:, 1]) # column 1

# Linear algebra
A = np.array([[1,2],[3,4]])
print(np.linalg.det(A)) # determinant
print(np.dot(A, A)) # matrix multiplication
```

Output

```
[ 2 4 6 8 10]
3.0 1.4142135623730951
(3, 3)
[[1 4 7]
[2 5 8]
[3 6 9]]
6
[2 5 8]
-2.0
[[ 7 10]
[15 22]]
```

11.2 Pandas

Pandas provides high-performance data structures (Series and DataFrame) for data manipulation and analysis. Think of a DataFrame as a powerful Excel spreadsheet with Python superpowers.

pandas_intro.py

```
import pandas as pd

# Create DataFrame
data = {
'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
'Age': [25, 30, 35, 28],
'Score': [88, 72, 95, 81],
'City': ['NY', 'LA', 'Chicago', 'NY']
}
df = pd.DataFrame(data)

print(df.head())
print(df.describe())

# Filtering
ny_students = df[df['City'] == 'NY']
print(ny_students)

high_score = df[df['Score'] > 80][['Name', 'Score']]
print(high_score)

# Aggregation
print(df.groupby('City')['Score'].mean())

# Adding columns
df['Grade'] = df['Score'].apply(lambda s: 'A' if s>=90 else ('B' if s>=80 else 'C'))
print(df)

# Read/Write CSV
# df.to_csv('students.csv', index=False)
# df = pd.read_csv('students.csv')
```

Output

```
Name Age Score City
0 Alice 25 88 NY
1 Bob 30 72 LA
2 Charlie 35 95 Chicago
3 Diana 28 81 NY

Age Score
count 4.0 4.000000
mean 29.5 84.000000
std 4.3 10.016653
min 25.0 72.000000
```

```
max 35.0 95.000000
```

■ Real-World Use Case

Pandas is the #1 tool for data cleaning, transformation, and exploration. It's used by data analysts at every major company to process sales data, customer records, financial reports, and survey results before feeding into ML models.

Chapter 12

Exception Handling

An **exception** is a runtime error that disrupts the normal flow of a program. Python uses try-except blocks to catch and handle these errors gracefully instead of crashing.

12.1 try-except

Exception	Cause
ZeroDivisionError	Division by zero
TypeError	Wrong type (e.g., 'a' + 1)
ValueError	Invalid value (e.g., int('abc'))
FileNotFoundError	File doesn't exist
IndexError	List index out of range
KeyError	Dict key not found
AttributeError	Object missing attribute
ImportError	Module not found

exceptions.py

```
# Full try-except-else-finally structure
def safe_divide(a, b):
try:
result = a / b
except ZeroDivisionError:
print("Error: Cannot divide by zero!")
return None
except TypeError as e:
print(f"Type error: {e}")
return None
else:
print("Division successful!")
return result
finally:
print("Operation complete.\n")

print(safe_divide(10, 2))
print(safe_divide(10, 0))
print(safe_divide(10, 'a'))

# Multiple exception handling
```

```
try:
data = [1, 2, 3]
print(data[10]) # IndexError
except (IndexError, KeyError) as e:
print(f"Access error: {e}")
```

■ Output

```
Division successful!
Operation complete.
5.0
Error: Cannot divide by zero!
Operation complete.
None
Type error: unsupported operand type(s) for /: 'int' and 'str'
Operation complete.
None
Access error: list index out of range
```

12.2 Custom Exceptions

You can define your own exception classes by inheriting from the Exception base class.

■ custom_exceptions.py

```
class ValidationError(Exception):
"""Raised when input validation fails."""
def __init__(self, field, message):
self.field = field
self.message = message
super().__init__(f"[{field}] {message}")

class AgeError(ValidationError):
pass

def register_user(name, age):
if not name or not name.strip():
raise ValidationError("name", "Name cannot be empty.")
if not isinstance(age, int) or age < 0 or age > 150:
raise AgeError("age", f"Invalid age: {age}. Must be 0-150.")
return f"User '{name}' (age {age}) registered!"

try:
print(register_user("Alice", 25))
print(register_user("", 25))
except ValidationError as e:
print(f"Validation failed: {e}")

try:
```

```
    print(register_user("Bob", -5))
except AgeError as e:
    print(f"Age error: {e}")
```

■ Output

```
User 'Alice' (age 25) registered!
Validation failed: [name] Name cannot be empty.
Age error: [age] Invalid age: -5. Must be 0-150.
```

■ Real-World Use Case

Custom exceptions are used in REST APIs (InvalidTokenError, PaymentFailedError), database layers (RecordNotFoundError), and financial apps (InsufficientFundsError) to provide meaningful error messages and enable structured error handling.

Chapter 13

Real-World Projects

Applying Python concepts to real projects is the best way to solidify your learning. Below are three complete, practical mini-projects covering different skill areas.

Project 1: Student Grade Manager

A console application that manages student records, computes averages, assigns grades, and saves/loads data from a JSON file. Demonstrates: dictionaries, file I/O, functions, exception handling, and list comprehensions.

grade_manager.py

```
import json
import os

DATA_FILE = "grades.json"

def load_data():
if os.path.exists(DATA_FILE):
with open(DATA_FILE) as f:
return json.load(f)
return {}

def save_data(data):
with open(DATA_FILE, 'w') as f:
json.dump(data, f, indent=2)

def add_student(data, name, scores):
data[name] = scores
save_data(data)
print(f"Added {name}.")

def get_grade(avg):
if avg >= 90: return 'A'
elif avg >= 80: return 'B'
elif avg >= 70: return 'C'
elif avg >= 60: return 'D'
return 'F'

def show_report(data):
print(f"\n{'Name':<15} {'Average':>8} {'Grade':>6}")
print('-' * 32)
for name, scores in data.items():
avg = sum(scores) / len(scores)
grade = get_grade(avg)
print(f"{name:<15} {avg:>8.1f} {grade:>6}")
```

```
print()

# Demo
db = load_data()
add_student(db, "Alice", [92, 88, 95, 91])
add_student(db, "Bob", [75, 68, 80, 72])
add_student(db, "Charlie", [55, 60, 58, 62])
show_report(db)
```

■ Output

```
Added Alice.
Added Bob.
Added Charlie.

Name Average Grade
--------------------------------
Alice 91.5 A
Bob 73.8 C
Charlie 58.8 F
```

Project 2: Simple To-Do List App

A class-based task manager with add, complete, delete, and list functionality, with persistence via JSON. Demonstrates: OOP, file handling, custom exceptions, and datetime.

■ todo_app.py

```
import json
import datetime

class TaskNotFoundError(Exception):
pass

class TodoApp:
def __init__(self, filename="tasks.json"):
self.filename = filename
self.tasks = self._load()

def _load(self):
try:
with open(self.filename) as f:
return json.load(f)
except FileNotFoundError:
return []

def _save(self):
with open(self.filename, 'w') as f:
json.dump(self.tasks, f, indent=2)

def add(self, title, priority="medium"):
```

```
    task = {
    "id": len(self.tasks) + 1,
    "title": title,
    "priority": priority,
    "done": False,
    "created": str(datetime.date.today())
    }
    self.tasks.append(task)
    self._save()
    print(f"Task #{task['id']} added: {title}")

    def complete(self, task_id):
    for task in self.tasks:
    if task['id'] == task_id:
    task['done'] = True
    self._save()
    print(f"Task #{task_id} marked complete.")
    return
    raise TaskNotFoundError(f"Task #{task_id} not found.")

    def list_tasks(self):
    print(f"\n{'ID':<5}{'Status':<10}{'Priority':<10} Title")
    print('-' * 45)
    for t in self.tasks:
    status = '[x]' if t['done'] else '[ ]'
    print(f"{t['id']:<5}{status:<10}{t['priority']:<10} {t['title']}")
    print()

    # Demo
    app = TodoApp()
    app.add("Learn Python decorators", "high")
    app.add("Build Flask API", "high")
    app.add("Read NumPy docs", "low")
    app.complete(1)
    app.list_tasks()
```

■ Output

```
Task #1 added: Learn Python decorators
Task #2 added: Build Flask API
Task #3 added: Read NumPy docs
Task #1 marked complete.

ID Status Priority Title
---------------------------------------------
1 [x] high Learn Python decorators
2 [ ] high Build Flask API
```

```
3 [ ] low Read NumPy docs
```

Project 3: Word Frequency Analyzer

Reads a text file, cleans and tokenizes it, computes word frequencies, and displays a ranked summary. Demonstrates: file I/O, string methods, dictionaries, sorting, list comprehensions, and generators.

word_analyzer.py

```
import re
from collections import Counter

def tokenize(text):
"""Lowercase and extract words only."""
return re.findall(r'\b[a-z]+\b', text.lower())

def word_frequency(filepath):
stop_words = {'the','a','an','and','or','but','in',
'on','at','to','for','is','it','of','with'}
try:
with open(filepath, 'r') as f:
text = f.read()
except FileNotFoundError:
return {}

words = [w for w in tokenize(text) if w not in stop_words]
return Counter(words)

def display_top(counter, n=10):
print(f"\n{'Rank':<6}{'Word':<20}{'Count':>6} Bar")
print('-' * 50)
for rank, (word, count) in enumerate(counter.most_common(n), 1):
bar = '#' * min(count, 20)
print(f"{rank:<6}{word:<20}{count:>6} {bar}")

# Demo with sample text
sample = """Python is a powerful programming language.
Python is used for web development, data science,
artificial intelligence and automation. Python developers
enjoy Python's clean syntax and python libraries."""

with open("sample.txt", "w") as f:
f.write(sample)

freq = word_frequency("sample.txt")
display_top(freq, 8)
```

Output

```
Rank Word Count Bar
--------------------------------------------------
```

```
1 python 4 ####
2 development 1 #
3 data 1 #
4 science 1 #
5 artificial 1 #
6 intelligence 1 #
7 automation 1 #
8 developers 1 #
```

What's Next?

Congratulations on completing this Python guide! To continue your journey: (1) Build a REST API with Flask or FastAPI. (2) Explore machine learning with scikit-learn. (3) Contribute to open-source Python projects on GitHub. (4) Practice on LeetCode, HackerRank, or Kaggle competitions. The best way to master Python is to build things — start today!

Concept	One-liner
List comprehension	`[x*2 for x in range(5)]`
Dict comprehension	`{k:v for k,v in pairs}`
Lambda	`f = lambda x: x**2`
Decorator	`@functools.wraps(fn)`
Generator	`yield value`
Context manager	`with open(f) as fh:`
Ternary	`x if condition else y`
Unpacking	`a, *b, c = [1,2,3,4,5]`
f-string	`f"Hello {name!r}"`
Walrus (:=)	`if (n := len(a)) > 10:`
Type hint	`def f(x: int) -> str:`