Easy Python

A Practical Guide to Learning Python

With runnable examples and output

Table of Contents

Chapter 1: Python Basics

Welcome to Python! This chapter covers the fundamental building blocks of the Python programming language.

Hello World

Your First Python Program

The 'print()' function displays text to the screen. It's the simplest way to output information.

```
print("Hello, World!")
print("Welcome to Easy Python!")
```

Output:

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

Variables and Data Types

Basic Variables

Variables store data. Python automatically determines the type based on the value assigned.

```
# Integer
age = 25
print(f"Age: {age}, Type: {type(age).__name__}")

# Float
price = 19.99
print(f"Price: {price}, Type: {type(price).__nam

# String
name = "Python"
print(f"Name: {name}, Type: {type(name).__name__

# Boolean
is_active = True
print(f"Active: {is_active}, Type: {type(is_acti
```

Output:

```
Age: 25, Type: int
Price: 19.99, Type: float
Name: Python, Type: str
Active: True, Type: bool
```

Type Conversion

Convert between types using 'int()', 'float()', 'str()', and 'bool()'.

```
# String to integer
num_str = "42"
num_int = int(num_str)
print(f"'{num_str}' -> {num_int} (type: {type(nu

# Integer to float
x = 10
y = float(x)
print(f"{x} -> {y}")

# Number to string
pi = 3.14159
pi_str = str(pi)
print(f"String version: '{pi_str}'")
```

Output:

```
'42' -> 42 (type: int)
10 -> 10.0
String version: '3.14159'
```

Basic Operators

Arithmetic Operators

Python supports all standard arithmetic operations plus floor division and exponentiation.

```
a, b = 17, 5

print(f"{a} + {b} = {a + b}")   # Addition
print(f"{a} - {b} = {a - b}")   # Subtraction
print(f"{a} * {b} = {a * b}")   # Multiplication
print(f"{a} / {b} = {a / b}")   # Division (floa
print(f"{a} // {b} = {a // b}") # Floor division
print(f"{a} % {b} = {a % b}")   # Modulo (remain
print(f"{a} ** {b} = {a ** b}") # Exponentiation
```

Output:

```
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3.4
17 // 5 = 3
17 % 5 = 2
17 ** 5 = 1419857
```

Comparison Operators

Comparison operators return boolean values ('True' or 'False').

```
x, y = 10, 20

print(f"{x} == {y}: {x == y}")  # Equal
print(f"{x} != {y}: {x != y}")  # Not equal
print(f"{x} < {y}: {x < y}")    # Less than
print(f"{x} > {y}: {x > y}")    # Greater than
print(f"{x} <= {y}: {x <= y}")  # Less or equal
print(f"{x} >= {y}: {x >= y}")  # Greater or equ
```

Output:

```
10 == 20: False
10 != 20: True
10 < 20: True
10 > 20: False
10 <= 20: True
10 >= 20: False
```

Input and Output

Formatted Output

F-strings (formatted string literals) provide a clean way to embed expressions in strings.

```
name = "Alice"
age = 30
height = 1.75

# F-string formatting
print(f"Name: {name}")
print(f"Age: {age} years old")
print(f"Height: {height:.2f} meters")
print(f"In 5 years, {name} will be {age + 5}")
```

Output:

```
Name: Alice
Age: 30 years old
Height: 1.75 meters
In 5 years, Alice will be 35
```

Chapter 2: Control Flow

Control flow statements allow your program to make decisions and repeat actions based on conditions.

Conditional Statements

If-Elif-Else

Use 'if', 'elif', and 'else' to execute different code blocks based on conditions.

```
score = 85

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"Score: {score} -> Grade: {grade}")
```

Output:

```
Score: 85 -> Grade: B
```

Logical Operators

Combine conditions using 'and', 'or', and 'not'.

```
age = 25
has_license = True
has_car = False

can_drive = age >= 18 and has_license
print(f"Can drive: {can_drive}")

needs_transport = not has_car
print(f"Needs transport: {needs_transport}")

can_travel = has_car or has_license
print(f"Can travel: {can_travel}")
```

Output:

```
Can drive: True
Needs transport: True
Can travel: True
```

Loops

For Loop

The 'for' loop iterates over sequences like lists, strings, or ranges.

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

print()

# Loop with range
for i in range(1, 6):
    print(f"Count: {i}")
```

Output:

```
I like apple
I like banana
I like cherry

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

While Loop

The 'while' loop repeats as long as a condition is true.

```
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1

print("Loop finished!")
```

Output:

```
Count is 0
Count is 1
Count is 2
Count is 3
Count is 4
Loop finished!
```

Loop Control

Use 'break' to exit a loop and 'continue' to skip to the next iteration.

```
# Break example
print("Finding first even number:")
for n in [1, 3, 5, 6, 7, 8]:
    if n % 2 == 0:
        print(f"Found: {n}")
        break

print()

# Continue example
print("Odd numbers only:")
for n in range(1, 8):
    if n % 2 == 0:
        continue
    print(n, end=" ")
print()
```

Output:

```
Finding first even number:
Found: 6

Odd numbers only:
1 3 5 7
```

Comprehensions

List Comprehension

Create lists concisely using list comprehension syntax.

```
# Traditional way
squares_old = []
for x in range(1, 6):
    squares_old.append(x ** 2)

# List comprehension
squares = [x ** 2 for x in range(1, 6)]
print(f"Squares: {squares}")

# With condition
evens = [x for x in range(10) if x % 2 == 0]
print(f"Evens: {evens}")
```

Output:

```
Squares: [1, 4, 9, 16, 25]
Evens: [0, 2, 4, 6, 8]
```

Dict and Set Comprehension

Similar syntax works for dictionaries and sets.

```
# Dictionary comprehension
word = "hello"
char_positions = {char: i for i, char in enumera
print(f"Character positions: {char_positions}")

# Set comprehension (unique values)
numbers = [1, 2, 2, 3, 3, 3, 4]
unique_squares = {x ** 2 for x in numbers}
print(f"Unique squares: {unique_squares}")
```

Output:

```
Character positions: {'h': 0, 'e': 1, 'l': 3, 'o
Unique squares: {16, 1, 4, 9}
```

Chapter 3: Data Structures

Python provides powerful built-in data structures for organizing and manipulating collections of data.

Lists

List Operations

Lists are ordered, mutable collections that can hold any type of data.

```
# Creating lists
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]

# Accessing elements
print(f"First: {numbers[0]}, Last: {numbers[-1]}

# Modifying
numbers.append(6)
numbers.insert(0, 0)
print(f"After append and insert: {numbers}")

# Slicing
print(f"Slice [1:4]: {numbers[1:4]}")
print(f"Every 2nd: {numbers[::2]}")
```

Output:

```
First: 1, Last: 5
After append and insert: [0, 1, 2, 3, 4, 5, 6]
Slice [1:4]: [1, 2, 3]
Every 2nd: [0, 2, 4, 6]
```

List Methods

Lists have many useful built-in methods.

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

# Sorting
fruits.sort()
print(f"Sorted: {fruits}")

# Counting and finding
print(f"Count 'apple': {fruits.count('apple')}")
print(f"Index of 'cherry': {fruits.index('cherry

# Removing
fruits.remove("apple")  # Removes first occurren
print(f"After remove: {fruits}")

last = fruits.pop()
print(f"Popped: {last}, Remaining: {fruits}")
```

Output:

```
Sorted: ['apple', 'apple', 'banana', 'cherry']
Count 'apple': 2
Index of 'cherry': 3
After remove: ['apple', 'banana', 'cherry']
Popped: cherry, Remaining: ['apple', 'banana']
```

Tuples

Working with Tuples

Tuples are immutable sequences, often used for fixed collections of items.

```
# Creating tuples
point = (10, 20)
person = ("Alice", 30, "Engineer")

# Accessing
print(f"Point: x={point[0]}, y={point[1]}")

# Unpacking
name, age, job = person
print(f"{name} is {age} years old, works as {job

# Tuples as return values
def get_min_max(numbers):
    return min(numbers), max(numbers)

data = [5, 2, 8, 1, 9]
minimum, maximum = get_min_max(data)
print(f"Min: {minimum}, Max: {maximum}")
```

Output:

```
Point: x=10, y=20
Alice is 30 years old, works as Engineer
Min: 1, Max: 9
```

Dictionaries

Dictionary Basics

Dictionaries store key-value pairs for fast lookups.

```
# Creating a dictionary
person = {
    "name": "Bob",
    "age": 25,
    "city": "New York"
}

# Accessing values
print(f"Name: {person['name']}")
print(f"Age: {person.get('age')}")

# Adding and updating
person["email"] = "bob@example.com"
person["age"] = 26
print(f"Updated: {person}")

# Safe access with default
country = person.get("country", "Unknown")
print(f"Country: {country}")
```

Output:

```
Name: Bob
Age: 25
Updated: {'name': 'Bob', 'age': 26, 'city': 'New
Country: Unknown
```

Dictionary Iteration

Iterate over keys, values, or both.

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

# Keys
print("Students:", list(scores.keys()))

# Values
print("Scores:", list(scores.values()))

# Both
print("\nAll entries:")
for name, score in scores.items():
    print(f"  {name}: {score}")
```

Output:

```
Students: ['Alice', 'Bob', 'Charlie']
Scores: [95, 87, 92]

All entries:
  Alice: 95
  Bob: 87
  Charlie: 92
```

Sets

Set Operations

Sets are unordered collections of unique elements, perfect for membership testing.

```
# Creating sets
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

print(f"Set A: {a}")
print(f"Set B: {b}")

# Set operations
print(f"Union (A | B): {a | b}")
print(f"Intersection (A & B): {a & b}")
print(f"Difference (A - B): {a - b}")
print(f"Symmetric diff (A ^ B): {a ^ b}")

# Membership
print(f"3 in A: {3 in a}")
print(f"7 in A: {7 in a}")
```

Output:

```
Set A: {1, 2, 3, 4}
Set B: {3, 4, 5, 6}
Union (A | B): {1, 2, 3, 4, 5, 6}
Intersection (A & B): {3, 4}
Difference (A - B): {1, 2}
Symmetric diff (A ^ B): {1, 2, 5, 6}
3 in A: True
7 in A: False
```

Chapter 4: Functions

Functions allow you to organize code into reusable blocks, making programs more modular and maintainable.

Defining Functions

Basic Functions

Use 'def' to define a function with parameters and a return value.

```
def greet(name):
    """Return a greeting message."""
    return f"Hello, {name}!"

def add(a, b):
    """Add two numbers."""
    return a + b

# Using functions
message = greet("Python")
print(message)

result = add(5, 3)
print(f"5 + 3 = {result}")
```

Output:

```
Hello, Python!
5 + 3 = 8
```

Default and Keyword Arguments

Parameters can have default values, and arguments can be passed by name.

```
def make_coffee(size="medium", milk=False, sugar
    """Describe a coffee order."""
    order = f"{size} coffee"
    if milk:
        order += " with milk"
    if sugar > 0:
        order += f" and {sugar} sugar(s)"
    return order

# Different ways to call
print(make_coffee())
print(make_coffee("large"))
print(make_coffee(milk=True, sugar=2))
print(make_coffee("small", True, 1))
```

Output:

```
medium coffee
large coffee
medium coffee with milk and 2 sugar(s)
small coffee with milk and 1 sugar(s)
```

Advanced Parameters

*args and **kwargs

Accept variable numbers of positional and keyword arguments.

```
def sum_all(*args):
    """Sum any number of arguments."""
    return sum(args)

def print_info(**kwargs):
    """Print key-value pairs."""
    for key, value in kwargs.items():
        print(f"  {key}: {value}")

# Using *args
print(f"Sum: {sum_all(1, 2, 3, 4, 5)}")

# Using **kwargs
print("Person info:")
print_info(name="Alice", age=30, city="Boston")
```

Output:

```
Sum: 15
Person info:
  name: Alice
  age: 30
  city: Boston
```

Lambda Functions

Anonymous Functions

Lambda functions are small, one-line anonymous functions.

```
# Simple lambda
square = lambda x: x ** 2
print(f"Square of 5: {square(5)}")

# Lambda with multiple arguments
add = lambda a, b: a + b
print(f"3 + 4 = {add(3, 4)}")

# Common use: sorting
students = [("Alice", 85), ("Bob", 92), ("Charli
students.sort(key=lambda x: x[1], reverse=True)
print("Sorted by score:")
for name, score in students:
    print(f"  {name}: {score}")
```

Output:

```
Square of 5: 25
3 + 4 = 7
Sorted by score:
  Bob: 92
  Alice: 85
  Charlie: 78
```

Scope

Variable Scope

Variables have different scopes: local, enclosing, global, and built-in (LEGB rule).

```
global_var = "I'm global"

def outer():
    enclosing_var = "I'm enclosing"

    def inner():
        local_var = "I'm local"
        print(local_var)
        print(enclosing_var)
        print(global_var)

    inner()

outer()

# Modifying global variable
counter = 0

def increment():
    global counter
    counter += 1

increment()
increment()
```

```
print(f"Counter: {counter}")
```

Output:

```
I'm local
I'm enclosing
I'm global
Counter: 2
```

Chapter 5: Modules and Pac

Modules help organize code into separate files. Python's standard library provides many useful modules.

Importing Modules

Import Syntax

Use 'import' to load modules and access their functions.

```
import math
from datetime import datetime, timedelta
from random import randint, choice

# Using math module
print(f"Pi: {math.pi:.4f}")
print(f"Square root of 16: {math.sqrt(16)}")
print(f"Ceiling of 4.2: {math.ceil(4.2)}")

# Using datetime
now = datetime.now()
print(f"Current time: {now.strftime('%Y-%m-%d %H

# Using random
print(f"Random number 1-10: {randint(1, 10)}")
print(f"Random choice: {choice(['apple', 'banana
```

Output:

```
Pi: 3.1416
Square root of 16: 4.0
Ceiling of 4.2: 5
Current time: 2026-02-01 22:03
Random number 1-10: 1
Random choice: banana
```

Standard Library Highlights

Collections Module

The collections module provides specialized container datatypes.

```
from collections import Counter, defaultdict, na

# Counter - count elements
words = ["apple", "banana", "apple", "cherry", "
count = Counter(words)
print(f"Word counts: {dict(count)}")
print(f"Most common: {count.most_common(2)}")

# defaultdict - dict with default values
grouped = defaultdict(list)
for word in words:
    grouped[len(word)].append(word)
print(f"Grouped by length: {dict(grouped)}")

# namedtuple - tuple with named fields
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(f"Point: x={p.x}, y={p.y}")
```

Output:

```
Word counts: {'apple': 3, 'banana': 2, 'cherry':
Most common: [('apple', 3), ('banana', 2)]
Grouped by length: {5: ['apple', 'apple', 'apple
Point: x=10, y=20
```

Itertools Module

Itertools provides efficient iterators for common patterns.

```
from itertools import count, cycle, islice, comb

# combinations
items = ['A', 'B', 'C']
print("Combinations of 2:")
for combo in combinations(items, 2):
    print(f"  {combo}")

# permutations
print("Permutations of 2:")
for perm in permutations(items, 2):
    print(f"  {perm}")

# islice - slice an iterator
from itertools import accumulate
numbers = [1, 2, 3, 4, 5]
running_sum = list(accumulate(numbers))
print(f"Running sum: {running_sum}")
```

Output:

```
Combinations of 2:
  ('A', 'B')
  ('A', 'C')
  ('B', 'C')
Permutations of 2:
  ('A', 'B')
  ('A', 'C')
  ('B', 'A')
  ('B', 'C')
  ('C', 'A')
  ('C', 'B')
Running sum: [1, 3, 6, 10, 15]
```

Chapter 6: Object-Oriented P

OOP allows you to model real-world entities using classes and objects, promoting code reuse and organization.

Classes and Objects

Defining a Class

A class is a blueprint for creating objects with attributes and methods.

```
class Dog:
    """A simple Dog class."""

    # Class attribute (shared by all instances)
    species = "Canis familiaris"

    def __init__(self, name, age):
        """Initialize a new Dog."""
        self.name = name  # Instance attribute
        self.age = age

    def bark(self):
        """Make the dog bark."""
        return f"{self.name} says Woof!"

    def describe(self):
        """Describe the dog."""
        return f"{self.name} is {self.age} years

# Creating objects
buddy = Dog("Buddy", 3)
max_dog = Dog("Max", 5)

print(buddy.describe())
```

```
print(buddy.bark())
print(f"Species: {buddy.species}")
```

Output:

```
Buddy is 3 years old
Buddy says Woof!
Species: Canis familiaris
```

Inheritance

Extending Classes

Inheritance allows a class to inherit attributes and methods from a parent class.

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

    def speak(self):
        raise NotImplementedError("Subclass must

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

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

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

# Polymorphism in action
animals = [Cat("Whiskers"), Dog("Buddy"), Cow("B
for animal in animals:
    print(animal.speak())
```

Output:

```
Whiskers says Meow!
Buddy says Woof!
Bessie says Moo!
```

Special Methods

Dunder Methods

Special methods (dunder methods) customize how objects behave with operators and built-in functions.

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

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

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

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y *

    def __abs__(self):
        return (self.x ** 2 + self.y ** 2) ** 0.

v1 = Vector(3, 4)
v2 = Vector(1, 2)

print(f"v1 = {v1}")
print(f"v2 = {v2}")
print(f"v1 + v2 = {v1 + v2}")
print(f"v1 * 3 = {v1 * 3}")
```

```
print(f"|v1| = {abs(v1)}")
```

Output:

```
v1 = Vector(3, 4)
v2 = Vector(1, 2)
v1 + v2 = Vector(4, 6)
v1 * 3 = Vector(9, 12)
|v1| = 5.0
```

Chapter 7: Error Handling

Proper error handling makes your programs robust and user-friendly by gracefully managing unexpected situations.

Try-Except

Basic Exception Handling

Use 'try-except' blocks to catch and handle errors.

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Error: Cannot divide by zero!"
    else:
        return f"{a} / {b} = {result}"
    finally:
        print("Division attempted.")

print(divide(10, 2))
print()
print(divide(10, 0))
```

Output:

```
Division attempted.
10 / 2 = 5.0

Division attempted.
Error: Cannot divide by zero!
```

Multiple Exceptions

Handle different types of exceptions differently.

```
def process_data(data, index):
    try:
        value = data[index]
        result = 100 / value
        return f"Result: {result}"
    except IndexError:
        return "Error: Index out of range"
    except ZeroDivisionError:
        return "Error: Division by zero"
    except TypeError:
        return "Error: Invalid data type"

data = [10, 0, 5, "abc"]

print(process_data(data, 0))
print(process_data(data, 1))
print(process_data(data, 10))
print(process_data(data, 3))
```

Output:

```
Result: 10.0
Error: Division by zero
Error: Index out of range
Error: Invalid data type
```

Raising Exceptions

Custom Exceptions

Raise exceptions to signal errors and create custom exception types.

```
class ValidationError(Exception):
    """Custom exception for validation errors.""
    pass

def validate_age(age):
    if not isinstance(age, int):
        raise TypeError("Age must be an integer"
    if age < 0:
        raise ValidationError("Age cannot be neg
    if age > 150:
        raise ValidationError("Age seems unreali
    return True

# Test validation
test_ages = [25, -5, 200, "thirty"]

for age in test_ages:
    try:
        validate_age(age)
        print(f"Age {age}: Valid")
    except (TypeError, ValidationError) as e:
        print(f"Age {age}: {e}")
```

Output:

```
Age 25: Valid
Age -5: Age cannot be negative
Age 200: Age seems unrealistic
Age thirty: Age must be an integer
```

Chapter 8: File and Data Har

Python provides simple yet powerful tools for reading, writing, and processing files and data.

Reading and Writing Files

Text Files

Use 'open()' with context managers to safely handle files.

```
import tempfile
import os

# Create a temporary file for demonstration
with tempfile.NamedTemporaryFile(mode='w', suffi
    temp_path = f.name
    f.write("Hello, Python!\n")
    f.write("This is line 2.\n")
    f.write("This is line 3.\n")

# Reading entire file
with open(temp_path, 'r') as f:
    content = f.read()
    print("Full content:")
    print(content)

# Reading line by line
print("Line by line:")
with open(temp_path, 'r') as f:
    for i, line in enumerate(f, 1):
        print(f"  Line {i}: {line.strip()}")

# Cleanup
os.unlink(temp_path)
```

Output:

```
Full content:
Hello, Python!
This is line 2.
This is line 3.

Line by line:
  Line 1: Hello, Python!
  Line 2: This is line 2.
  Line 3: This is line 3.
```

JSON Data

Working with JSON

JSON is a popular format for storing and exchanging data.

```
import json

# Python object to JSON
data = {
    "name": "Alice",
    "age": 30,
    "languages": ["Python", "JavaScript", "Go"],
    "active": True
}

# Convert to JSON string
json_str = json.dumps(data, indent=2)
print("JSON string:")
print(json_str)

print()

# Parse JSON back to Python
parsed = json.loads(json_str)
print(f"Name: {parsed['name']}")
print(f"Languages: {', '.join(parsed['languages'
```

Output:

```
JSON string:
{
  "name": "Alice",
  "age": 30,
  "languages": [
    "Python",
    "JavaScript",
    "Go"
  ],
  "active": true
}

Name: Alice
Languages: Python, JavaScript, Go
```

Path Handling

Using pathlib

The pathlib module provides an object-oriented approach to file paths.

```
from pathlib import Path

# Current directory
current = Path.cwd()
print(f"Current dir: {current.name}")

# Path operations
example_path = Path("/home/user/documents/report
print(f"Name: {example_path.name}")
print(f"Stem: {example_path.stem}")
print(f"Suffix: {example_path.suffix}")
print(f"Parent: {example_path.parent}")
print(f"Parts: {example_path.parts}")

# Building paths
new_path = Path("data") / "2024" / "january" / "
print(f"Built path: {new_path}")
```

Output:

```
Current dir: test-python-book
Name: report.pdf
Stem: report
Suffix: .pdf
Parent: /home/user/documents
Parts: ('/', 'home', 'user', 'documents', 'repor
Built path: data/2024/january/report.csv
```

Chapter 9: Advanced Python

These advanced features help you write more elegant, efficient, and Pythonic code.

Iterators

Custom Iterator

Iterators provide a way to access elements one at a time without loading all into memory.

```
class Countdown:
    """An iterator that counts down from n to 1.

    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= 0:
            raise StopIteration
        self.n -= 1
        return self.n + 1

# Using the iterator
print("Countdown from 5:")
for num in Countdown(5):
    print(num, end=" ")
print()
```

Output:

```
Countdown from 5:
5 4 3 2 1
```

Generators

Generator Functions

Generators use 'yield' to produce values lazily, saving memory.

```
def fibonacci(n):
    """Generate first n Fibonacci numbers."""
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

# Using the generator
print("First 10 Fibonacci numbers:")
for num in fibonacci(10):
    print(num, end=" ")
print()

# Generator expression
squares = (x**2 for x in range(1, 6))
print(f"Squares: {list(squares)}")
```

Output:

```
First 10 Fibonacci numbers:
0 1 1 2 3 5 8 13 21 34
Squares: [1, 4, 9, 16, 25]
```

Decorators

Function Decorators

Decorators wrap functions to add behavior without modifying the original function.

```
import time

def timer(func):
    """Decorator that prints execution time."""
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__} took {end - star
        return result
    return wrapper

def logger(func):
    """Decorator that logs function calls."""
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {ar
        return func(*args, **kwargs)
    return wrapper

@timer
@logger
def slow_add(a, b):
    time.sleep(0.01)  # Simulate slow operation
    return a + b
```

```
result = slow_add(3, 5)
print(f"Result: {result}")
```

Output:

```
Calling slow_add with (3, 5)
wrapper took 0.012515 seconds
Result: 8
```

Context Managers

Custom Context Manager

Context managers handle setup and cleanup using 'with' statements.

```
from contextlib import contextmanager

class Timer:
    """Context manager for timing code blocks.""

    def __enter__(self):
        import time
        self.start = time.perf_counter()
        return self

    def __exit__(self, *args):
        import time
        self.elapsed = time.perf_counter() - sel
        print(f"Elapsed time: {self.elapsed:.6f}

# Using the context manager
with Timer():
    total = sum(range(100000))
    print(f"Sum: {total}")

# Using contextmanager decorator
@contextmanager
def tag(name):
    print(f"<{name}>")
```

```
    yield
    print(f"</{name}>")

with tag("html"):
    with tag("body"):
        print("  Hello, World!")
```

Output:

```
Sum: 4999950000
Elapsed time: 0.000941 seconds
<html>
<body>
  Hello, World!
</body>
</html>
```

Chapter 10: Testing and Bes

Writing tests ensures your code works correctly and continues to work as you make changes.

Unit Testing

Writing Tests

Python's unittest module provides a framework for writing and running tests.

```
import unittest

def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero"
    return a / b

class TestMathFunctions(unittest.TestCase):
    def test_add_positive(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative(self):
        self.assertEqual(add(-1, -1), -2)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)
```

```
# Run tests
suite = unittest.TestLoader().loadTestsFromTestC
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

Type Hints

Adding Type Annotations

Type hints improve code readability and enable static type checking.

```
from typing import Optional

def greet(name: str) -> str:
    """Return a greeting message."""
    return f"Hello, {name}!"

def find_max(numbers: list[int]) -> Optional[int
    """Find the maximum number, or None if list
    if not numbers:
        return None
    return max(numbers)

def process_data(
    data: dict[str, int],
    multiplier: float = 1.0
) -> dict[str, float]:
    """Multiply all values by a multiplier."""
    return {k: v * multiplier for k, v in data.i

# Using the functions
print(greet("Python"))
print(f"Max of [1, 5, 3]: {find_max([1, 5, 3])}"
print(f"Max of []: {find_max([])}")
```

```
data = {"a": 10, "b": 20}
print(f"Processed: {process_data(data, 1.5)}")
```

Output:

```
Hello, Python!
Max of [1, 5, 3]: 5
Max of []: None
Processed: {'a': 15.0, 'b': 30.0}
```

Chapter 11: Concurrency an

Python offers multiple ways to handle concurrent operations: threading, multiprocessing, and async/await.

Threading

Basic Threading

Threads allow concurrent execution, useful for I/O-bound tasks.

```
import threading
import time

def worker(name, delay):
    """Simulate a worker task."""
    print(f"{name} starting")
    time.sleep(delay)
    print(f"{name} finished after {delay}s")

# Create threads
threads = [
    threading.Thread(target=worker, args=("Worke
    threading.Thread(target=worker, args=("Worke
    threading.Thread(target=worker, args=("Worke
]

# Start all threads
start = time.perf_counter()
for t in threads:
    t.start()

# Wait for all to complete
for t in threads:
    t.join()
```

```
elapsed = time.perf_counter() - start
print(f"All workers done in {elapsed:.2f}s")
```

Output:

```
Worker-1 starting
Worker-2 starting
Worker-3 starting
Worker-2 finished after 0.05s
Worker-3 finished after 0.08s
Worker-1 finished after 0.1s
All workers done in 0.10s
```

Async/Await

Async Programming

Async/await provides efficient handling of I/O-bound operations.

```
import asyncio

async def fetch_data(name, delay):
    """Simulate fetching data."""
    print(f"Fetching {name}...")
    await asyncio.sleep(delay)
    return f"{name} data"

async def main():
    # Run tasks concurrently
    tasks = [
        fetch_data("users", 0.1),
        fetch_data("products", 0.08),
        fetch_data("orders", 0.05),
    ]

    start = asyncio.get_event_loop().time()
    results = await asyncio.gather(*tasks)
    elapsed = asyncio.get_event_loop().time() -

    print(f"Results: {results}")
    print(f"Total time: {elapsed:.2f}s")

# Run the async main function
```

```
asyncio.run(main())
```

Output:

```
Fetching users...
Fetching products...
Fetching orders...
Results: ['users data', 'products data', 'orders
Total time: 0.10s
```

Chapter 12: Practical Applica

Let's apply what we've learned to build some practical, real-world examples.

Data Processing

Processing CSV-like Data

A common task is processing structured data.

```
# Sample data (normally from CSV)
sales_data = [
    {"product": "Widget", "quantity": 100, "pric
    {"product": "Gadget", "quantity": 50, "price
    {"product": "Widget", "quantity": 75, "price
    {"product": "Gizmo", "quantity": 30, "price"
    {"product": "Gadget", "quantity": 25, "price
]

# Calculate totals by product
from collections import defaultdict

totals = defaultdict(lambda: {"quantity": 0, "re

for item in sales_data:
    product = item["product"]
    totals[product]["quantity"] += item["quantit
    totals[product]["revenue"] += item["quantity

# Display results
print("Sales Summary:")
print("-" * 40)
for product, data in sorted(totals.items()):
    print(f"{product:10} | Qty: {data['quantity'
```

```
total_revenue = sum(d["revenue"] for d in totals
print("-" * 40)
print(f"{'Total':10} |            | ${total_reve
```

Output:

```
Sales Summary:
----------------------------------------
Gadget     | Qty:   75 | $1,874.25
Gizmo      | Qty:   30 | $1,499.70
Widget     | Qty:  175 | $1,748.25
----------------------------------------
Total      |            | $5,122.20
```

Simple API Client

Building a Simple Class-Based Client

Encapsulate functionality in a clean, reusable class.

```
from dataclasses import dataclass
from typing import Optional
from datetime import datetime

@dataclass
class Task:
    id: int
    title: str
    completed: bool = False
    created_at: datetime = None

    def __post_init__(self):
        if self.created_at is None:
            self.created_at = datetime.now()

class TaskManager:
    def __init__(self):
        self._tasks: dict[int, Task] = {}
        self._next_id = 1

    def add(self, title: str) -> Task:
        task = Task(id=self._next_id, title=titl
        self._tasks[self._next_id] = task
        self._next_id += 1
```

```
        return task

    def complete(self, task_id: int) -> Optional
        if task_id in self._tasks:
            self._tasks[task_id].completed = Tru
            return self._tasks[task_id]
        return None

    def list_all(self) -> list[Task]:
        return list(self._tasks.values())

# Using the TaskManager
manager = TaskManager()
manager.add("Learn Python basics")
manager.add("Practice with exercises")
manager.add("Build a project")

manager.complete(1)

print("Task List:")
for task in manager.list_all():
    status = "[x]" if task.completed else "[ ]"
    print(f"  {status} {task.id}. {task.title}")
```

Output:

```
Task List:
  [x] 1. Learn Python basics
  [ ] 2. Practice with exercises
  [ ] 3. Build a project
```