THE COMPLETE GUIDE

PYTHON PROGRAMMING

From Zero to Advanced

A completely self-contained, step-by-step teaching document covering every core concept of the Python language — from your very first **print()** statement to generators, decorators, async/await, OOP, and the broader Python ecosystem.
No prior programming experience required.

14 CHAPTERS · ILLUSTRATED · SELF-CONTAINED

Variables · Strings · Operators · Control Flow · Lists · Dicts · Sets
Functions · Modules · Files · OOP · Generators · Decorators · Async

>>>

Table of Contents

Fourteen chapters covering every major Python concept.

CHAPTER 01

01 Introduction to Python

History · Philosophy · How It Works · Setup · Syntax

What is Python?

Python is a **high-level, general-purpose, interpreted** programming language created by **Guido van Rossum**, first released in **1991**. Named after Monty Python, the language was designed to prioritise *readability* and *simplicity*. Its guiding philosophy is captured in **PEP 20 — The Zen of Python**: *'Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex.'* Python is open-source and stewarded by the **Python Software Foundation (PSF)**.

Why Python? Real-World Use Cases

Domain	What Python powers
`Web Dev`	Django, Flask, FastAPI — build websites and REST APIs rapidly.
`Data Science`	Pandas, NumPy, Matplotlib — the lingua franca of data analysis.
`AI / ML`	TensorFlow, PyTorch, scikit-learn are all Python-first ecosystems.
`Automation`	File operations, browser tasks, OS workflows scripted in minutes.
`DevOps/Cloud`	Ansible, AWS Boto3, GCP client libraries — all Python.
`Science`	SciPy, SymPy used in academia and research worldwide.

How Python Works: Interpreter & Bytecode

Python is an **interpreted** language. When you run a **.py** file, **CPython** (the standard interpreter) compiles it to platform-independent **bytecode** (.pyc files stored in __pycache__). That bytecode is then executed by the **Python Virtual Machine (PVM)**. This two-step process is invisible to you but explains why Python starts slightly slower than compiled languages while running consistently across every platform.

Installation and Environment

Download from **python.org**. On Windows, tick *'Add Python to PATH'*. Verify with python --version. Recommended editors: **VS Code** (free, lightweight, great Python extension), **PyCharm** (full IDE, Community edition is free), **IDLE** (ships with Python — good for absolute beginners).

Your First Program

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

# Output:
Hello, World!
```

Python Syntax Fundamentals

Indentation defines code blocks — Python uses 4 spaces per level (not braces). Inconsistent indentation raises IndentationError. **Case-sensitive**: myVar and myvar are different. **Comments**: # for single-line; triple-quoted strings as docstrings.

```
# Single-line comment
if True:
    print('Indented — belongs to the if')  # 4 spaces
print('Not indented — always runs')

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

✓ Run **import this** in any Python interpreter to read the Zen of Python — 19 aphorisms that define the Python philosophy.

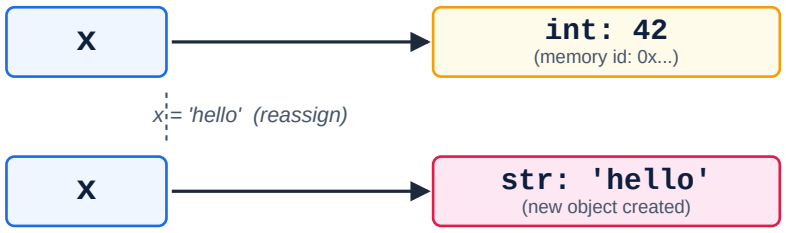
CHAPTER 02

02 Variables and Data Types

Dynamic Typing · Core Types · Conversion · Memory Model

Variables: Dynamic Typing

Variables are **named labels** pointing to objects in memory — not typed boxes. Python infers the type automatically at assignment. The same name can reference an integer, then a string, then a list. Use type() to inspect, isinstance() to check. id() returns the object's memory address.



Variables are NAMES,
not boxes containing values.

Reassigning does NOT mutate the first object — it points the name to a brand-new object in memory.

Figure 2.1 — Python's memory model: names point to objects.

```
x = 42
print(type(x), id(x))   # <class 'int'>  140234567
x = 'hello'             # x now points to a str object
print(isinstance(True, int))   # True  — bool is a subclass of int!
```

Core Data Types

Type	Description & example
`int`	Whole numbers, unlimited size: 42, -7, 10**100
`float`	IEEE-754 double precision: 3.14 (0.1+0.2 != 0.3 exactly!)
`complex`	Real + imaginary: 3+4j. Access: c.real, c.imag
`str`	Immutable Unicode sequence: 'hello'. Indexable, sliceable
`bool`	True (==1) / False (==0). Subclass of int
`NoneType`	Single value None — absence of a value. Check with: x is None

Type Conversion

Python does **not** automatically convert between types. Use explicit constructors. Raises ValueError if incompatible.

```
int('42')        # 42
float('3.14')    # 3.14
str(100)         # '100'
bool(0)          # False
bool('hello')    # True    — non-empty string is truthy
int('hello')     # ValueError!
```

! **Falsy values:** False, 0, 0.0, '', [], {}, (), None — everything else is truthy.

CHAPTER 03

03 Strings in Depth

Indexing · Slicing · Methods · f-strings · Immutability

Creating Strings

Strings can use **single**, **double**, or **triple** quotes. Single and double are interchangeable. Triple-quoted strings span multiple lines.

```
s1 = 'Hello'
s2 = "World"
s3 = '''multi-line
string'''
greeting = s1 + ', ' + s2   # concatenation
line     = '-' * 40          # repetition
```

Indexing and Slicing

Characters are indexed from **0** (left) and **-1** (right). Slice syntax: s[start:stop:step] — stop is *exclusive*.

```
s = 'Python'
#    P  y  t  h  o  n
#    0  1  2  3  4  5   (positive)
#   -6 -5 -4 -3 -2 -1   (negative)

s[0]      # 'P'
s[-1]     # 'n'
s[1:4]    # 'yth'
s[::-1]   # 'nohtyP'  (reverse)
s[::2]    # 'Pto'      (every 2nd char)
```

Essential String Methods

Method	What it does
`upper() / lower()`	Change case: 'hello'.upper() -> 'HELLO'

`strip()`	Remove leading/trailing whitespace
`split(sep)`	'a,b,c'.split(',') -> ['a','b','c']
`join(iterable)`	'-'.join(['a','b']) -> 'a-b'
`replace(old,new)`	'hello'.replace('l','r') -> 'herro'
`find(sub)`	First index or -1: 'hello'.find('ll') -> 2
`startswith/endswith()`	Boolean prefix/suffix check
`count(sub)`	Count occurrences: 'hello'.count('l') -> 2
`zfill(width)`	'42'.zfill(5) -> '00042'
`format() / f-string`	Embed expressions (see below)

f-strings — Modern String Formatting

Prefix a string with f and embed any expression inside {}. Supports format specs: {value:.2f} for two decimal places.

```
name  = 'Alice'
score = 95.678
print(f'Hello, {name}!')           # Hello, Alice!
print(f'Score: {score:.2f}')        # Score: 95.68
print(f'10^2 = {10**2}')            # 10^2 = 100
print(f'{name!r}')                  # 'Alice'
```

Escape Characters & Raw Strings

```
path = r'C:\Users\Alice'   # raw — backslash is literal
print('Line1\nLine2')      # \n = newline
print('Tab\there')         # \t = tab
```

> ℹ Strings are **immutable** — every method returns a NEW string. Original is unchanged.

CHAPTER 04

04 Operators and Expressions

Arithmetic · Comparison · Logical · Bitwise · Precedence

Arithmetic Operators

```
a, b = 17, 5
a + b    # 22   addition
a - b    # 12   subtraction
a * b    # 85   multiplication
a / b    # 3.4  true division  (always float)
a // b   # 3    floor division (integer quotient)
a % b    # 2    modulo (remainder)
a ** b   # 1419857  exponentiation
```

> ! Floor division rounds toward **negative infinity**: -7 // 2 = -4, not -3. Modulo is widely used for even/odd tests: n % 2 == 0.

Comparison and Logical Operators

```
# Comparison — always returns bool
5 == 5     # True
5 != 3     # True
0 < 5 < 10 # True  — chained!

# Logical — short-circuit
True and False    # False
True or  False    # True
not True          # False
```

Bitwise Operators

```
a, b = 0b1010, 0b1100   # 10 and 12 in binary
a & b    # 8    AND:  0b1000
a | b    # 14   OR:   0b1110
a ^ b    # 6    XOR:  0b0110
~a       # -11  NOT  (inverts bits)
a << 1   # 20   left shift  (* 2)
a >> 1   # 5    right shift (// 2)
```

Operator Precedence (Highest to Lowest)

Operator(s)	Description
`**`	Exponentiation
`+x -x ~x`	Unary plus, minus, bitwise NOT
`* / // %`	Multiplication, division, floor, modulo
`+ -`	Addition, subtraction
`<< >>`	Bit shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`== != < > is in`	Comparisons, identity, membership
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR (lowest precedence)

CHAPTER 05

05 Control Flow

if/elif/else · Ternary · while · for · break/continue/pass

if / elif / else

Conditions are evaluated top-to-bottom. The **first truthy** branch executes; the rest are skipped. else catches everything that didn't match.

```
score = 72
if score >= 90:   grade = 'A'
elif score >= 80: grade = 'B'
elif score >= 70: grade = 'C'
else:             grade = 'F'

# Ternary expression (one-liner)
label = 'even' if score % 2 == 0 else 'odd'
```

while Loops

Repeats as long as the condition is True. break exits immediately. continue skips to the next iteration. A while...else block runs only if the loop completed without hitting a break.

```
count = 0
while count < 5:
    print(count, end=' ')   # 0 1 2 3 4
    count += 1

# Infinite loop with break
while True:
    cmd = input('> ')
    if cmd == 'quit': break
```

for Loops

Python's for iterates over any **iterable**. range(start, stop, step) generates integers lazily. enumerate() provides index + value. zip() pairs elements from multiple iterables.

```
for i in range(5):              # 0 1 2 3 4
    print(i, end=' ')

fruits = ['apple', 'banana', 'cherry']
for i, fruit in enumerate(fruits, start=1):
    print(f'{i}. {fruit}')        # 1. apple ...

names  = ['Alice', 'Bob']
scores = [95, 87]
for name, score in zip(names, scores):
    print(f'{name}: {score}')
```

pass is a no-op placeholder — use it when a block is syntactically required but you want nothing to happen yet.

CHAPTER 06

06 Lists and Tuples

Mutability · Methods · Comprehensions · Packing · namedtuple

Lists — Ordered & Mutable

Defined with []. Supports indexing, slicing, in-place modification.

```
nums = [10, 20, 30, 40, 50]
nums[0]      # 10
nums[-1]     # 50
nums[1:3]    # [20, 30]
nums[2] = 99 # mutation
```

Essential List Methods

Method	What it does
`append(x)`	Add x to end
`extend(iter)`	Add all items from iterable
`insert(i, x)`	Insert x at index i
`remove(x)`	Remove first occurrence (ValueError if absent)
`pop(i=-1)`	Remove and return item at index i
`sort(key=None)`	Sort in place. key= accepts a function
`reverse()`	Reverse in place
`copy()`	Shallow copy
`index(x)`	Return index of first occurrence
`count(x)`	Count occurrences

List Comprehensions

Syntax: [expression for item in iterable if condition]. Faster than equivalent for-loops because of bytecode optimisation.

```
squares = [x**2 for x in range(10)]
# [0,1,4,9,16,25,36,49,64,81]

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

# Flatten a 2D list
flat = [n for row in [[1,2],[3,4]] for n in row]
```

Tuples — Immutable

Like lists but immutable. Faster, hashable, signal 'do not change'. Single-element needs trailing comma: (42,).

```
point = (3, 4)             # packing
x, y  = point              # unpacking
first, *rest = [1, 2, 3, 4, 5]
a, b = b, a                # elegant swap

from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(3, 4)
p.x, p.y                   # 3  4
```

CHAPTER 07

07 Dictionaries and Sets

Key-Value Pairs · Methods · Comprehensions · Set Algebra

Dictionaries

An **ordered* mapping** of unique hashable keys to values. Python 3.7+ preserves insertion order. O(1) average access via hashing.

```
person = {'name': 'Alice', 'age': 30}
person['name']            # 'Alice'
person.get('salary', 0)   # 0 (safe, no KeyError)
person['age'] = 31        # update
person['job'] = 'Eng'     # add key
del person['age']         # remove key

for k, v in person.items():
    print(f'{k}: {v}')
```

Dictionary Methods

Method	What it does
`keys()`	View of all keys
`values()`	View of all values
`items()`	View of (key, value) pairs
`get(k,default)`	Safe access — no KeyError
`setdefault(k,v)`	Set if missing, return value
`update(other)`	Merge another dict in
`pop(k)`	Remove and return value

Sets — Unique, Unordered

O(1) membership test. {1,2,3} or set(iter). **Note:** {} creates an empty *dict* — use set().

```
a = {1,2,3,4,5}; b = {3,4,5,6,7}
a | b   # Union:        {1,2,3,4,5,6,7}
a & b   # Intersection: {3,4,5}
a - b   # Difference:   {1,2}
a ^ b   # Symmetric:    {1,2,6,7}

# Remove duplicates from a list
list(set([1,2,2,3,3,3]))  # [1,2,3]
```

frozenset is an immutable set — hashable, so it can be a dict key or set element. Create with frozenset(iter).

CHAPTER 08

08 Functions

*args · **kwargs · LEGB Scope · Lambda · Recursion · Hints

Defining Functions

Functions are **first-class objects**. Without return the function returns None. Multiple return values are packed into a tuple.

```
def add(a, b):
    """Return the sum."""
    return a + b

def min_max(nums):
    return min(nums), max(nums)   # tuple

lo, hi = min_max([5,2,9,1])  # lo=1, hi=9
```

Parameters: Default, *args, **kwargs

```
def greet(name, greeting='Hello'):
    return f'{greeting}, {name}!'
greet('Alice')              # Hello, Alice!
greet('Bob', 'Hi')          # Hi, Bob!

def total(*args):           # collects into tuple
    return sum(args)
total(1, 2, 3, 4)           # 10

def show(**kwargs):         # collects into dict
    for k,v in kwargs.items(): print(k,v)
show(name='Alice', age=30)
```

Scope: The LEGB Rule

Name resolution: **L**ocal → **E**nclosing → **G**lobal → **B**uilt-in. Python checks each scope in order — the first match wins.

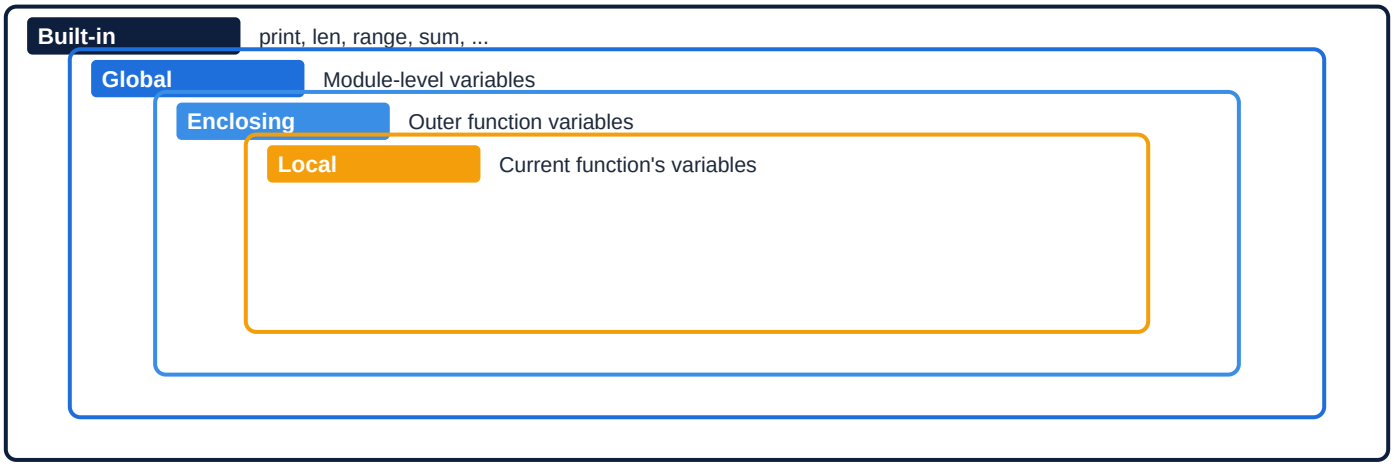


Figure 8.1 — LEGB: Python searches scopes from inside outward.

Lambda · map · filter · reduce

```
double = lambda x: x * 2

nums = [1, 2, 3, 4, 5]
list(map(lambda x: x*2,       nums))  # [2,4,6,8,10]
list(filter(lambda x: x%2==0, nums))  # [2,4]

from functools import reduce
reduce(lambda a,b: a*b, nums)         # 120
```

Recursion

```
def factorial(n):
    if n == 0: return 1          # base case
    return n * factorial(n - 1)  # recursive case

factorial(5)   # 120
```

CHAPTER 09

09 Modules and Packages

import · Standard Library · pip · Virtual Environments

Importing Modules

```
import math                    # access as math.sqrt()
from math import sqrt, pi      # specific names
import numpy as np             # alias

math.sqrt(16)  # 4.0
sqrt(25)       # 5.0
```

__name__ == '__main__'

When run *directly*: __name__ == '__main__'. When *imported*: __name__ equals the module name. This guard lets you write code that only runs when executed directly.

```
# utils.py
def useful(): return 42

if __name__ == '__main__':
    print(useful())  # only when run directly
```

Standard Library Highlights

Module	What it provides
`os / pathlib`	Filesystem: paths, dirs, env vars (OOP paths in pathlib)
`sys`	Runtime: sys.argv, sys.exit(), sys.path
`math`	sqrt, floor, ceil, log, sin, cos, pi, e, factorial
`random`	random(), randint(), choice(), shuffle(), sample()
`datetime`	datetime.now(), timedelta, strftime/strptime
`json`	dumps()/loads() — Python <-> JSON string
`re`	Regex: match(), search(), findall(), sub()
`collections`	Counter, defaultdict, deque, OrderedDict, namedtuple
`itertools`	chain, islice, product, combinations, cycle
`functools`	reduce, partial, lru_cache, wraps

pip and Virtual Environments

pip installs from PyPI: pip install requests. Bulk: pip install -r requirements.txt. Export current packages: pip freeze > requirements.txt.

A **virtual environment** isolates packages per project. Create: python -m venv .venv. Activate: .venv\Scripts\activate (Windows) or source .venv/bin/activate (Mac/Linux).

CHAPTER 10

10 File Handling and Exceptions

open() · Read/Write · CSV/JSON · try/except/finally

Opening Files

Always use the **with** statement — guarantees the file is closed even on exceptions. Modes: r read, w write/truncate, a append, rb/wb binary.

```
with open('data.txt', 'r', encoding='utf-8') as f:
    text  = f.read()        # whole file
    lines = f.readlines()   # list of lines
    for line in f:          # memory-efficient
        print(line.strip())

with open('out.txt', 'w', encoding='utf-8') as f:
    f.write('Hello\n')
```

CSV and JSON

```
import csv, json

with open('data.csv', newline='') as f:
    for row in csv.DictReader(f):
        print(row['name'], row['age'])

data     = {'scores': [95, 87, 92]}
as_str   = json.dumps(data, indent=2)
restored = json.loads(as_str)
```

Exception Handling

```
try:
    n = int(input('Number: '))
    r = 100 / n
except ValueError:
    print('Not a valid integer.')
except ZeroDivisionError:
    print('Cannot divide by zero.')
except Exception as e:
    print(f'Unexpected: {e}')
else:
    print(f'Result: {r}')  # no exception
finally:
    print('Done.')        # always runs
```

Custom Exceptions

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        super().__init__(f'Need {amount}, have {balance}')
        self.balance = balance
        self.amount  = amount

def withdraw(bal, amt):
    if amt > bal:
        raise InsufficientFundsError(bal, amt)
    return bal - amt
```

CHAPTER 11

11 Object-Oriented Programming

Classes · Inheritance · Dunder Methods · @property

Classes and Objects

__init__ is the constructor. self refers to the current instance. **Class attributes** are shared; **instance attributes** are per-object.

```
class Dog:
    species = 'Canis familiaris'   # class attr

    def __init__(self, name, age):
        self.name = name           # instance attrs
        self.age  = age

    def bark(self):
        return f'{self.name} says Woof!'

    def __str__(self):
        return f'Dog({self.name}, age={self.age})'

rex = Dog('Rex', 3)
print(rex.bark())   # Rex says Woof!
```

Inheritance & Polymorphism

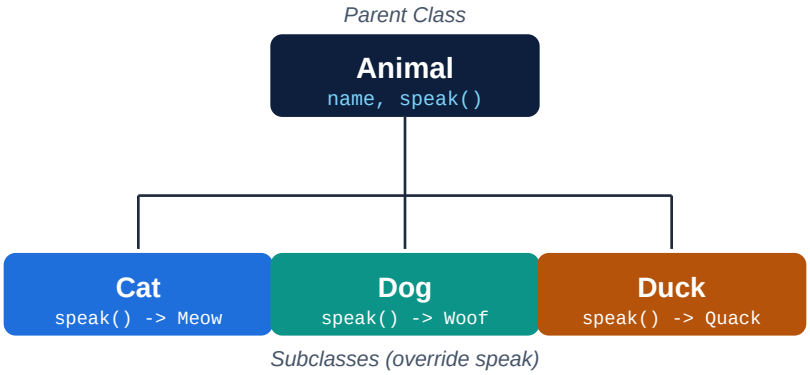


Figure 11.1 — Animal is the parent class; Cat, Dog, Duck override speak().

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

class Cat(Animal):
    def speak(self): return 'Meow'

class Duck(Animal):
    def speak(self): return 'Quack'

for a in [Cat('K'), Duck('D')]:
    print(a.name, a.speak())  # polymorphic call
```

Dunder (Magic) Methods

Method	Triggered by
`__init__`	Constructor
`__str__`	str(obj) and print(obj)
`__repr__`	Developer repr()
`__len__`	len(obj)
`__eq__`	== operator
`__lt__`	< operator (enables sort)
`__add__`	+ operator overloading
`__contains__`	in operator
`__iter__`	Makes object iterable
`__enter__/__exit__`	Context manager (with statement)

@property — Computed Attributes

```
class Temperature:
    def __init__(self, c): self._c = c
    @property
    def celsius(self): return self._c
    @celsius.setter
    def celsius(self, v):
        if v < -273.15: raise ValueError
        self._c = v
    @property
    def fahrenheit(self): return self._c*9/5+32

t = Temperature(100)
t.fahrenheit  # 212.0
```

CHAPTER 12

12 Advanced Python Concepts

Iterators · Generators · Decorators · collections · itertools

Iterators and Iterables

An **iterable** implements __iter__(). An **iterator** also implements __next__() and raises StopIteration when exhausted. The for loop calls both automatically.

```
class CountUp:
    def __init__(self, n): self.n=n; self.i=0
    def __iter__(self): return self
    def __next__(self):
        if self.i >= self.n: raise StopIteration
        self.i += 1; return self.i

for x in CountUp(3): print(x)   # 1  2  3
```

Generators — Lazy Sequences

Use yield instead of return. Execution resumes after each yield. **Lazy**: values are produced on demand — no memory overhead.

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
[next(fib) for _ in range(8)]   # [0,1,1,2,3,5,8,13]

# ~8 MB vs ~200 bytes:
big_list = [x**2 for x in range(1_000_000)]
big_gen  = (x**2 for x in range(1_000_000))
```

Decorators

Wraps a function to add behaviour. @decorator = func = decorator(func). Use @functools.wraps to preserve metadata.

```
import functools, time

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        t0 = time.perf_counter()
        r  = func(*args, **kwargs)
        print(f'{func.__name__}: {time.perf_counter()-t0:.4f}s')
        return r
    return wrapper

@timer
def slow_sum(n): return sum(range(n))
slow_sum(1_000_000)
```

collections & itertools

Tool	What it does
`Counter(iter)`	Count elements: {'a':3,'b':2}
`defaultdict(list)`	Auto-creates missing keys
`deque`	O(1) append/pop from both ends
`chain(*iters)`	Iterate multiple iters as one
`islice(iter, n)`	First n items lazily
`product(a, b)`	Cartesian product
`combinations(it, r)`	r-length combos no repeat
`permutations(it, r)`	All r-length orderings

CHAPTER 13

13 Functional Programming & Concurrency

Closures · Partial · Threading · Multiprocessing · asyncio

Closures

Inner function captures variables from its enclosing scope even after that scope exits. Underpins decorators and stateful functions without classes.

```
def make_multiplier(factor):
    def multiplier(x):    # closes over factor
        return x * factor
    return multiplier

double = make_multiplier(2)
triple = make_multiplier(3)
double(5), triple(5)   # 10, 15
```

Concurrency: Three Approaches

Python offers three concurrency models. Choose based on your bottleneck:

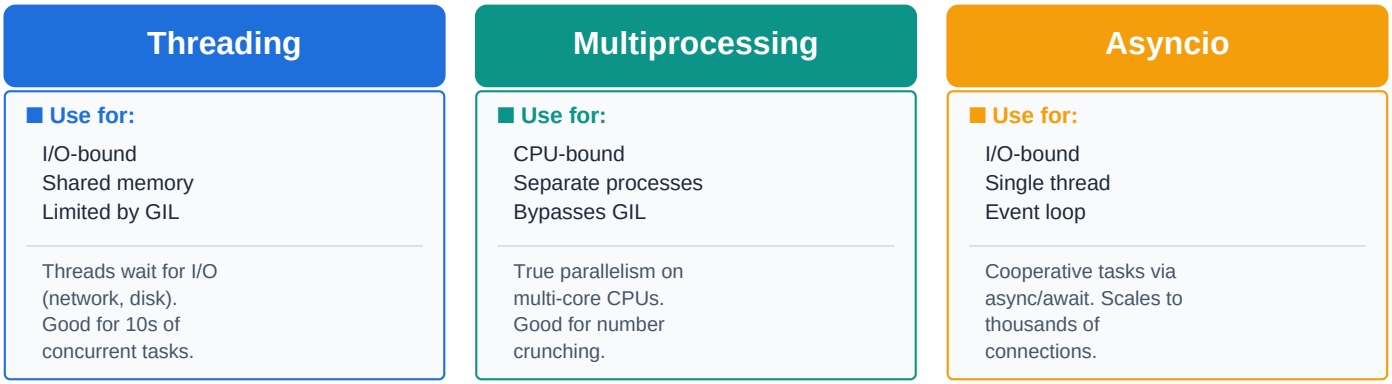


Figure 13.1 — Threading, multiprocessing, and asyncio compared.

asyncio Example

```
import asyncio

async def fetch(url):
    await asyncio.sleep(1)   # simulate I/O
    return f'Data from {url}'

async def main():
    results = await asyncio.gather(
        fetch('http://api1.com'),
        fetch('http://api2.com'),
    )
    print(results)

asyncio.run(main())   # ~1s total (concurrent), not 2s
```

✓ **Quick guide:** Use **asyncio** for high-concurrency I/O. Use **threading** for simpler I/O concurrency. Use **multiprocessing** for CPU-bound work that must run in parallel.

CHAPTER 14

14 Best Practices & The Python Ecosystem

PEP 8 · Pythonic Code · Type Hints · Testing · Libraries

PEP 8 Style Guide

Rule	Guideline
`Indentation`	4 spaces per level. Never mix tabs and spaces.
`Line length`	Max 79 chars. Use () for implicit continuation.
`Blank lines`	2 between top-level defs; 1 between methods.
`Imports`	One per line. Group: stdlib -> 3rd party -> local.

Spaces	Around operators, after commas. Not inside brackets.
Naming	snake_case: vars/funcs. PascalCase: classes. ALL_CAPS: constants.
Tools	flake8 (lint), black (auto-format), isort (sort imports).

Pythonic Code: EAFP vs LBYL

```
# LBYL — Look Before You Leap (less Pythonic)
if 'key' in d: value = d['key']

# EAFP — Easier to Ask Forgiveness (Pythonic)
try:
    value = d['key']
except KeyError:
    value = default

# Best for dicts: use .get()
value = d.get('key', default)
```

Type Hints (PEP 484)

```
def greet(name: str, times: int = 1) -> str:
    return (name + '! ') * times

from typing import Optional, List, Dict
def find_user(uid: int) -> Optional[str]: ...
```

Key Third-Party Libraries

Library	Purpose
NumPy	N-D arrays, vectorised math — scientific foundation
Pandas	DataFrames for data analysis; CSV/Excel/SQL/JSON
Matplotlib	2D plotting and visualisation
Requests	Human-friendly HTTP: GET, POST, sessions, auth
Flask	Lightweight web framework / REST API
Django	Full-stack: ORM, admin, auth, templating
FastAPI	Async-first, auto OpenAPI docs, very fast
SQLAlchemy	ORM and SQL toolkit
PyTorch	Dynamic deep learning — preferred for AI research
TensorFlow	Google's ML platform — train and deploy

The Zen of Python — import this

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Readability counts.
Errors should never pass silently.
Now is better than never.
If the implementation is hard to explain, it's a bad idea.
```

✓ **Where to go next?** Official docs at **docs.python.org** are comprehensive and free. Build real projects: a web scraper, REST API, data pipeline, CLI tool. Read others' code on GitHub. The fastest path to mastery is writing lots of Python.

```
>>>
```

FIN

```
print('Thank you for reading.')
Thank you for reading.
```

Happy coding · docs.python.org