GateXAIML

AIML | Data Science | CSE | Mathematics

GATE

Data Science and Artificial Intelligence

Python Programming

A Complete Guide...

- Comprehensive Coverage
- GATE Focused
- Python Fundamentals to Advanced
- Practice Questions

Contents

About the Book

Artificial Intelligence and Machine Learning (AI/ML) are transforming industries across the globe — from healthcare and finance to transportation and education. From medical diagnosis systems and fraud detection to personalized recommendations and autonomous vehicles, AI/ML is shaping the way we live, work, and interact with technology.

To support this rapidly growing field, the GATE Data Science and Artificial Intelligence (DA) exam was introduced as a national-level gateway to higher studies, research, and employment opportunities in top institutions and organizations. The exam tests a candidate's proficiency in mathematics, programming, data handling, machine learning, and AI fundamentals.

This book is a compact and comprehensive guide for GATE DA aspirants. It is designed to help learners build a strong conceptual foundation while developing the problem-solving skills required for the exam. Many solved examples are included to illustrate key concepts, and each chapter features carefully crafted problems for practice.

Solutions to selected problems and topic-wise lectures will be discussed in detail on my YouTube channel (@GATEX-Aiml). All the concepts covered in the book will also be taught step-by-step through video tutorials, making this a complete learning resource for GATE DA preparation.

This book is designed for aspirants of the GATE DA exam focusing on **Python Programming**. It systematically covers theory, solved examples, and practice problems aligned with the official syllabus.

Dedicated to all my Gurus and Students.
"Knowledge grows only when shared — and it must remain free, for that is how it thrives."

STOP!

Attention!

Some examples solved in video lectures are different from those given in this book.

The procedure to solve problems and examples is well explained in the video lectures, and it is highly recommended to go through the video lectures for complete understanding.

Official Video Playlist

Watch on YouTube

Chapter 1

Python Basics

1.1 Language Overview

1.1.1 Python Characteristics

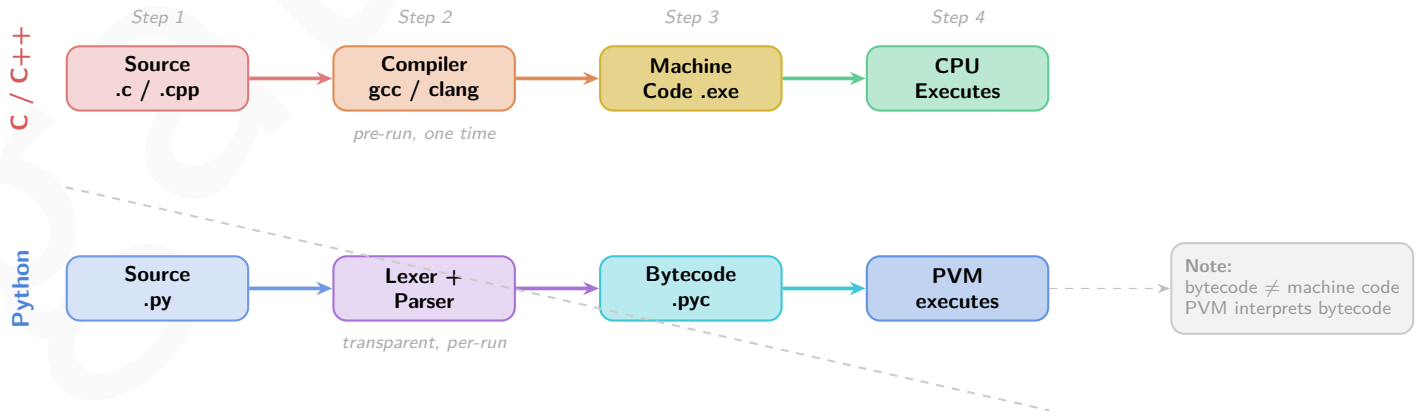
1.1.1.1 Interpreted Language

Interpreted Language — Definition

A programming language is called **interpreted** when its source code is executed *directly* by an interpreter program at runtime, statement by statement, without a prior separate compilation step that produces machine code. Python uses a **two-phase** execution model:

1. **Compilation phase** (automatic, transparent): source `.py` is compiled to platform-independent *bytecode* (`.pyc`) and cached in `__pycache__/`
2. **Interpretation phase**: the **Python Virtual Machine (PVM)** reads bytecode instructions one by one and executes them

This is why Python is called interpreted: *no explicit compile step is needed by the programmer*, and errors surface at runtime.



Properties of Python's Interpreted Model

- **No explicit compile step** — run with `python script.py` directly
- **Bytecode caching** — `__pycache__/script.cpython-3xx.pyc` created automatically; re-used if source unchanged (checked via modification timestamp)
- **Platform independence** — same `.py` runs on Windows, Linux, macOS (bytecode is *not* platform-specific)
- **Late error detection** — syntax errors caught before run; *runtime* errors (TypeError, NameError ...) only when that line executes
- **Dynamic code execution** — `eval()` and `exec()` compile and run strings as Python code at runtime
- **Interactive REPL** — possible because lines are interpreted immediately

Example 1: Runtime vs Compile-time Error Detection

```
# SyntaxError is caught BEFORE execution (static analysis)
# def bad(:           # SyntaxError -- Python detects before line 1 runs

# NameError only caught when the faulty line actually executes
def process(items):
    result = []
    for item in items:
        result.append(item * 2)    # line 4: runs fine
    return undefined_var            # line 5: NameError raised HERE

process([1, 2, 3])    # lines 1-4 execute successfully, crash on line 5
# NameError: name 'undefined_var' is not defined
```

Takeaway: Python executes code up to the point of failure. In a compiled language such code would be caught before any execution.

Example 2: Inspecting Bytecode with `dis`

```
import dis

def multiply(a, b):
    return a * b

dis.dis(multiply)
# Bytecode output (Python 3.11+):
#   3           RESUME          0
#   4           LOAD_FAST       0 (a)
#               LOAD_FAST       1 (b)
#               BINARY_OP       5 (*)
#               RETURN_VALUE
```

The `dis` module reveals the actual bytecode instructions the PVM executes. Each `LOAD_FAST` pushes a local variable onto the evaluation stack; `BINARY_OP` pops two operands, multiplies them, and pushes the result.

Example 3: Dynamic Execution with `eval` and `exec`

```
# eval: evaluates an expression string, returns result
expr = "3 ** 4 + len('hello')"
result = eval(expr)
print(result)    # 86

# exec: executes a block of Python code from a string
code = """
def greet(name):
    return f"Hello, {name}!"

```

```
message = greet("GATE")
"""
namespace = {}
exec(code, namespace)
print(namespace["message"])    # Hello, GATE!

# Practical: dynamic formula evaluator
formula = input("Enter formula: ")    # user types: 2*x + 3
x = 10
print(eval(formula))    # 23
```

`eval`/`exec` are possible *only* because Python is interpreted: bytecode is compiled and run at runtime from a string.

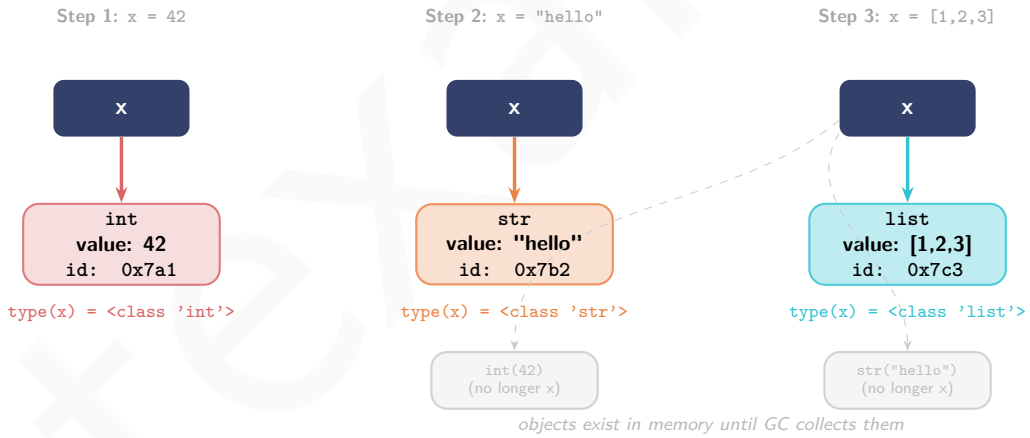
1.1.1.2 Dynamically Typed

Dynamic Typing — Definition

In a **dynamically typed** language, *type information is associated with values (objects), not with variables*. A variable is merely a **label** (name binding) that can be rebound to any object of any type at any time.
Formally: Type checking is performed at *runtime* (not compile time). The type of a variable is determined by the type of the object it currently refers to.
Contrast with static typing (C, Java, Rust):

Static (C/Java) `int x = 5;` — `x` is *permanently* an integer

Dynamic (Python) `x = 5` — `x` is a name; right now it points to `int(5)`



Properties of Dynamic Typing in Python

- **Variable has no type** — `x = 5` then `x = "hi"` is perfectly legal
- **Object has a type** — every object carries type info: `type(42)` → `<class 'int'>`
- **Type checking at runtime** — `TypeError` raised when an *operation* is attempted on an incompatible type
- **Duck typing** — "if it walks like a duck and quacks like a duck..." — Python checks *capabilities*, not declared types
- **Optional static type hints** (PEP 484) — `x: int = 5` is a *hint* only; Python itself does **not** enforce it at runtime (tools like `mypy` do)
- **Identity vs equality** — `id(obj)` returns unique identity; `is` checks identity, == checks value equality

Example 4: Variable Rebinding and `type()` / `id()`

```python
x = 42
print(type(x), id(x))    # <class 'int'>  9789376

x = "hello"
print(type(x), id(x))    # <class 'str'>  140223847823984

x = [1, 2, 3]
print(type(x), id(x))    # <class 'list'> 140223847701248

x = lambda n: n**2
print(type(x), id(x))    # <class 'function'> 140223846982144

# id changes every time -- x now refers to a different object
# The variable 'x' itself has no type; its current referent does
```

Example 5: Duck Typing in Practice

```python
# Any object with __len__ works -- no type declaration needed
def print_length(obj):
    print(f"Length: {len(obj)}")    # uses duck typing

print_length("Python")           # Length: 6  (str has __len__)
print_length([10, 20, 30])       # Length: 3  (list has __len__)
print_length({"a": 1, "b": 2})   # Length: 2  (dict has __len__)
# print_length(42)               # TypeError: 'int' has no len()

# Same function works for any "sized" object -- no isinstance() check!
import numpy as np
arr = np.array([1, 2, 3, 4])
print_length(arr)                # Length: 4
```

Example 6: Type Hints (PEP 484) — Optional Static Annotation

```python
# Type hints are ADVISORY ONLY -- Python does NOT enforce them
def add(a: int, b: int) -> int:
    return a + b

print(add(3, 4))        # 7   -- correct usage
print(add("x", "y"))    # xy  -- no runtime error! hint is ignored

# Runtime check (explicit) using isinstance
def safe_add(a: int, b: int) -> int:
    if not (isinstance(a, int) and isinstance(b, int)):
        raise TypeError(f"Expected int, got {type(a)}, {type(b)}")
    return a + b

safe_add("x", "y")
# TypeError: Expected int, got <class 'str'>, <class 'str'>
```

Use `mypy` or `pyright` externally to enforce type hints statically.

1.1.1.3 Strongly Typed

Strong Typing — Definition

A language is **strongly typed** when it does *not perform implicit type coercions* between unrelated types. Operations on incompatible types raise a `TypeError` rather than silently converting one operand.
Formal distinction:

Axis	Dynamic vs Static	Strong vs Weak
Concerns	*when* types are checked	*how strictly* types are enforced
Python	Dynamic (runtime)	Strong (no implicit coercion)
C	Static (compile time)	Weak (int/float coercions allowed)
JS	Dynamic (runtime)	Weak ("5"+2 = "52")

Python is **both** dynamically typed *and* strongly typed.

Python (Strong)	JavaScript (Weak)
"5"+2 → TypeError	"5"+2 → "52"
5+True → 6 (bool⊂int)	null+1 → 1
int("3")+2 → explicit	"3"-1 → 2 (implicit!)
1=="1" → False	1=="1" → false (==), true (===)

Strong Typing Rules in Python

- **No implicit string↔int coercion**: `"5" + 3` raises `TypeError`; must use `int("5") + 3`
- **Boolean is a subtype of int**: `True` behaves as 1 and `False` as 0 (this is the *only* notable implicit coercion)
- **Explicit conversion always required**: `str()`, `int()`, `float()`, `list()` etc. must be called manually
- **Comparison across types returns `False`**: `1 == "1"` is `False` (not a TypeError) but `1 < "1"` raises `TypeError`
- **+ for numbers means addition; for strings means concatenation**: mixing raises `TypeError`
- **Operator overloading is explicit**: classes define `__add__`, `__mul__` etc. to control behaviour

Example 7: Strong Typing — Implicit Coercion Blocked

```python
# ------ TypeError cases (strong typing) ------
try:
    result = "Age: " + 25          # can't concat str + int
except TypeError as e:
    print(e)   # can only concatenate str (not "int") to str

try:
    result = "3" * 1.5             # str * float not defined
except TypeError as e:
    print(e)   # can't multiply sequence by non-int of type 'float'

# ------ Explicit conversion (correct way) ------
result = "Age: " + str(25)         # "Age: 25"
result = int("3") * 2              # 6
result = float("3.14") + 1         # 4.14

# ------ bool IS int (one exception) ------
print(True + 1)        # 2   (True == 1)
print(False * 10)      # 0   (False == 0)
print(sum([True, True, False, True]))  # 3  (useful for counting)
```

Example 8: Cross-type Comparisons

```python
# == works across types (returns False, not error)
print(1 == "1")        # False
print(1 == 1.0)        # True  (int and float compare by value)
print(True == 1)       # True  (bool is subtype of int)
print(True == 1.0)     # True

# < / > raise TypeError for unrelated types (Python 3)
try:
    print(1 < "2")
except TypeError as e:
    print(e)   # '<' not supported between instances of 'int' and 'str'

# In Python 2, this used to silently return an arbitrary ordering!
# Python 3 fixed this -- another example of stronger type discipline.
```

1.1.1.4 Everything is an Object

Python's Unified Object Model

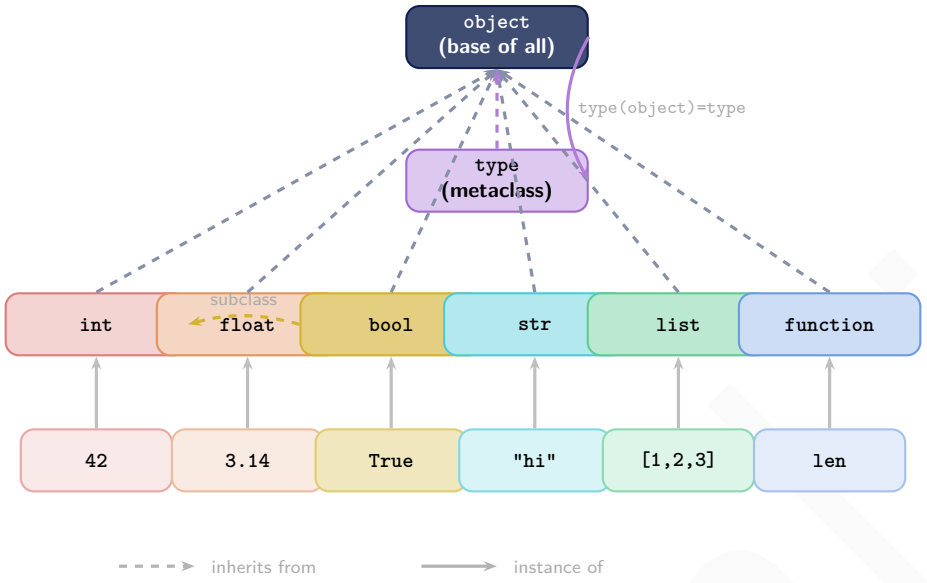
In Python, **every value is an object** — an instance of some class. This includes:

- Primitive values: `42`, `3.14`, `True`, `None`
- Collections: `list`, `tuple`, `dict`, `set`
- Functions and lambda expressions
- Classes themselves (instances of `type`)
- Modules and packages
- Frames, code objects, tracebacks

Every object has:

1. **Identity** — unique integer, returned by `id(obj)` (CPython: memory address)
2. **Type** — the class it belongs to, returned by `type(obj)`
3. **Value** — the data it holds

All objects ultimately inherit from the built-in class `object`.



First-Class Objects in Python

In Python, **first-class object** means an entity that can be:

- **Assigned to a variable**: `f = len`
- **Passed as an argument**: `map(str, [1,2,3])`
- **Returned from a function**: closures, decorators
- **Stored in a data structure**: `ops = [add, sub, mul]`
- **Has attributes and methods**: `f.__name__`, `(42).bit_length()`

This applies to *functions, classes, modules, and all built-in types*.

Example 9: Functions and Classes as Objects

```
# Functions are objects
def square(n):
    return n ** 2

print(type(square))           # <class 'function'>
print(square.__name__)        # square
print(square.__code__.co_varnames)  # ('n',)

% Assign function to a variable
f = square
print(f(7))                   # 49

% Store functions in a list
ops = [abs, str, type, id]
for op in ops:
    print(op(-5))             # 5 / '-5' / <class 'int'> / 9789280

% Classes are objects too (instances of 'type')
class Dog:
    sound = "Woof"

print(type(Dog))              # <class 'type'>
print(Dog.__bases__)          # (<class 'object'>,)
print(Dog.__dict__['sound']) # Woof

% Create a class dynamically using type()
Cat = type("Cat", (object,), {"sound": "Meow", "legs": 4})
```

```python
c = Cat()
print(c.sound, c.legs)        # Meow 4
```

Example 10: Integers and Strings Have Methods — They Are Objects

```python
# int methods
n = 255
print(n.bit_length())         # 8   (255 = 1111_1111)
print(n.to_bytes(2, 'big'))   # b'\x00\xff'
print((1_000_000).bit_count()) # 7 (Python 3.10+)

# float methods
x = 3.75
print(x.is_integer())         # False
print(x.as_integer_ratio())   # (15, 4)

# str methods
s = "gate exam"
print(s.upper())              # GATE EXAM
print(s.title())              # Gate Exam
print(s.count('a'))           # 2

# bool is int subclass
print(isinstance(True, int))  # True
```

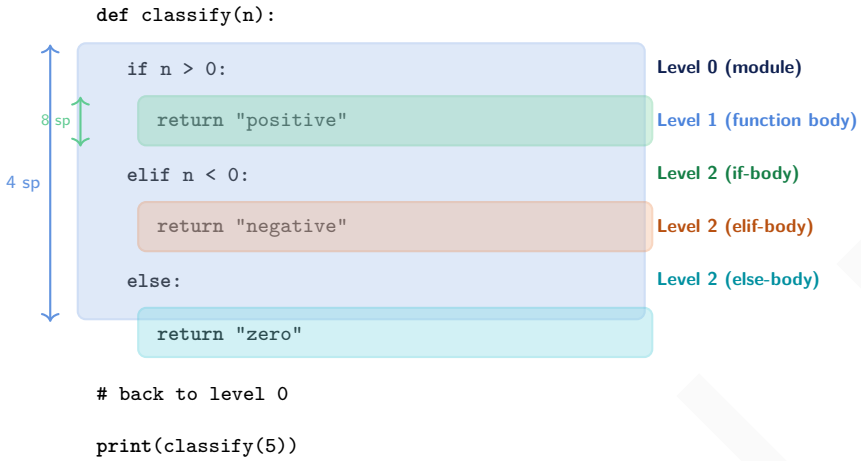
Example 11: Modules and None Are Objects

```python
import math

# Module is an object
print(type(math))               # <class 'module'>
print(math.__name__)            # math
print(math.__file__)            # /usr/lib/python3.x/lib-dynload/...
print(dir(math)[:5])            # ['__doc__', '__loader__', ...]

# None is a singleton object
print(type(None))               # <class 'NoneType'>
print(id(None) == id(None))     # True  (same singleton)

# Checking for None: use 'is', not '=='
result = None
if result is None:              # correct idiom
    print("No result yet")
```

1.1.1.5 Indentation Defines Blocks

Indentation as Block Structure

Python uses **consistent indentation** (whitespace at the start of a line) to delimit **code blocks**, replacing the braces {} used by C, Java, and many other languages.
Rule: All statements in a block must be indented by the *same number of spaces*. The block begins after a colon (:) on the preceding line and ends when indentation decreases back to the enclosing level.
Recommendation (PEP 8): use **4 spaces** per indentation level. *Never mix tabs and spaces* (raises `TabError` in Python 3).



Indentation Rules and Common Errors

- **Colon triggers a new block**: after `if`, `elif`, `else`, `for`, `while`, `def`, `class`, `with`, `try/except`
- **Consistent spacing required**: mixing 2-space and 4-space indentation inside the same block raises `IndentationError`
- **Tabs vs spaces**: Python 3 forbids mixing tabs and spaces (`TabError`)
- **`pass` placeholder**: used to create an empty block: `def todo(): pass`
- **Continuation lines**: long expressions can span multiple lines inside parentheses, brackets, or braces — indentation of continuation lines is flexible
- **One-liner blocks**: allowed but not recommended: `if x > 0: print(x)`

Example 12: Correct vs Incorrect Indentation

```python
# CORRECT: consistent 4-space blocks
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))    # 120

#            ERRORS


# IndentationError: unexpected indent
# def broken():
#     x = 1
#       y = 2      # <-- extra spaces, not a new block!

# IndentationError: expected an indented block
# def empty():
# print("hi")      # nothing indented after def

# TabError (Python 3): inconsistent use of tabs and spaces
# def mixed():
#     x = 1        # 4 spaces
#   y = 2      # 1 tab  -->  TabError!
```

Example 13: Nested Blocks and `pass`

```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Three levels of nesting
for i, row in enumerate(matrix):        # level 1 block
    for j, val in enumerate(row):       # level 2 block
        if val % 2 == 0:                # level 3 block
            print(f"[{i}][{j}] = {val} (even)")
        else:
            pass                        # deliberate no-op

# Output:
# [0][1] = 2 (even)
# [1][0] = 4 (even)
# [1][2] = 6 (even)
# [2][1] = 8 (even)
```

Example 14: Line Continuation Inside Parentheses

```python
# Long expressions: parentheses allow free indentation
result = (
    10
    + 20
    + 30
    + 40
)
print(result)   # 100

# Long function call
output = sorted(
    ['banana', 'apple', 'cherry', 'date'],
    key=len,
    reverse=True
)
print(output)   # ['banana', 'cherry', 'apple', 'date']

# Backslash continuation (avoid when possible)
total = 10 + \
        20 + \
        30
print(total)    # 60
```

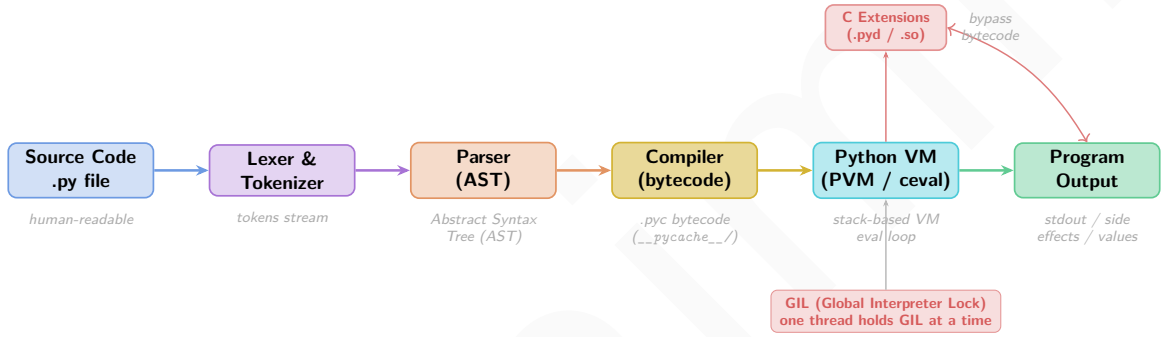
1.1.1.6 CPython: Reference Implementation & Execution Model

CPython — The Reference Implementation

CPython is the *official, reference implementation* of Python, written in C. When people say "Python", they almost always mean CPython.
Other implementations exist for specific use cases:

Implementation	Written In	Use Case
CPython	C	Default, most compatible
PyPy	Python (RPython)	JIT compilation, ∼5× faster
Jython	Java	JVM integration, call Java libraries
IronPython	C#	.NET / CLR integration
MicroPython	C	Microcontrollers (ESP32, Raspberry Pi Pico)
Brython	JavaScript	Python in the browser

All conforming implementations follow the **Python Language Reference** (`docs.python.org/3/reference`).



The Global Interpreter Lock (GIL)

The **GIL** is a mutex (lock) in CPython that allows only **one thread** to execute Python bytecode at a time, even on multi-core hardware.
Why it exists: CPython's memory management (reference counting) is not thread-safe. The GIL prevents race conditions by serialising access.
Consequences:

- **CPU-bound multithreading** gives no speedup (threads take turns holding the GIL)
- **I/O-bound multithreading** works well (GIL is released during I/O waits)
- **Multiprocessing** (`multiprocessing` module) bypasses GIL — each process has its own GIL
- **C extensions** can release the GIL manually (NumPy does this)

Python 3.13+: experimental "free-threaded" build (`-disable-gil`) is in progress under PEP 703.

CPython Memory Management Model

- **Reference Counting**: every object stores `ob_refcnt`; when count drops to zero the object is immediately deallocated
- **Cyclic GC**: a generational garbage collector handles *reference cycles* (objects referencing each other) that reference counting alone cannot reclaim
- **Small-integer cache**: CPython caches integers -5 to 256; `id(5) is id(5)` is always `True`
- **String interning**: short strings and identifiers are interned (one copy in memory); `sys.intern()` forces interning
- **Memory allocator**: CPython has its own arena/pool allocator (`obmalloc`) for objects ≤ 512 bytes

Example 15: Checking CPython Version and Implementation

```python
import sys
import platform

# Which implementation?
print(sys.implementation.name)  # cpython
print(sys.version)              # 3.11.5 (default, ...) [GCC 12.3.0]

# Detailed version info
print(sys.version_info)
# sys.version_info(major=3, minor=11, micro=5, ...)

# Platform
print(platform.python_implementation())  # CPython
print(platform.python_version())         # 3.11.5

# Check if we can use free-threaded mode (3.13+)
print(getattr(sys, '_is_gil_enabled', lambda: True)())  # True (GIL on)
```

Example 16: Reference Counting and sys.getrefcount

```python
import sys

x = []                          # ref count = 1 (x)
print(sys.getrefcount(x))       # 2 (x + getrefcount argument)

y = x                           # ref count = 2 (x and y)
z = x                           # ref count = 3
print(sys.getrefcount(x))       # 4 (x, y, z + getrefcount arg)

del y                           # ref count = 3
del z                           # ref count = 2
# When ref count reaches 0, object is immediately freed

# Small-int cache: same object for -5..256
a = 100; b = 100
print(a is b)     # True  (same cached object)
print(id(a) == id(b))  # True

a = 1000; b = 1000
print(a is b)     # False  (outside cache range, different objects)
```

Example 17: GIL Effect — CPU-bound Threads vs Processes

```python
import threading
import multiprocessing
import time

def count_down(n):
    while n > 0:
        n -= 1

N = 50_000_000

#         Single thread


t0 = time.perf_counter()
count_down(N)
print(f"Single  : {time.perf_counter()-t0:.2f}s")   # ~1.8s

#         Two threads (GIL prevents true parallelism)
t0 = time.perf_counter()
t1 = threading.Thread(target=count_down, args=(N//2,))
```

```python
t2 = threading.Thread(target=count_down, args=(N//2,))
t1.start(); t2.start(); t1.join(); t2.join()
print(f"Threads : {time.perf_counter()-t0:.2f}s")   # ~1.9s  (not faster!)

#         Two processes (each has own GIL)

t0 = time.perf_counter()
p1 = multiprocessing.Process(target=count_down, args=(N//2,))
p2 = multiprocessing.Process(target=count_down, args=(N//2,))
p1.start(); p2.start(); p1.join(); p2.join()
print(f"Processes: {time.perf_counter()-t0:.2f}s")  # ~0.9s  (2x faster!)
```

Conclusion: For CPU-bound tasks, use `multiprocessing` to escape the GIL. For I/O-bound tasks (network, disk), `threading` or `asyncio` is sufficient.

Example 18: The `ast` Module — Inspecting the Parse Tree

```python
import ast

source = """
x = 10
y = x * 2 + 1
print(y)
"""

tree = ast.parse(source)
print(ast.dump(tree, indent=2))
# Module(body=[
#   Assign(targets=[Name(id='x', ...)], value=Constant(value=10)),
#   Assign(targets=[Name(id='y', ...)],
#          value=BinOp(
#            left=BinOp(left=Name(id='x'), op=Mult(), right=Constant(2)),
#            op=Add(), right=Constant(1))),
#   Expr(value=Call(func=Name(id='print'), args=[Name(id='y')]))
# ])

# Walk the AST to find all function calls
for node in ast.walk(tree):
    if isinstance(node, ast.Call):
        if isinstance(node.func, ast.Name):
            print(f"Call: {node.func.id}")   # Call: print
```

This shows exactly what CPython's parser produces before the compiler converts the AST to bytecode instructions.

1.1.2 Running Python

1.1.2.1 Interactive Mode (REPL)

REPL — Read-Eval-Print Loop

REPL stands for **Read-Eval-Print Loop**. Python's interactive interpreter starts when you run `python3` (or `python`) with no arguments. Each expression you type is *read*, *evaluated*, and its result *printed* before the prompt reappears.

Key features:

- **Immediate feedback** — ideal for exploration and debugging
- **_ (underscore)** stores the last expression result
- `help(obj)` shows inline documentation
- `dir(obj)` lists all attributes/methods
- Enhanced REPLs: **IPython**, **Jupyter Notebook** (cells = mini-REPL)

- Exit with `exit()`, `quit()`, or `Ctrl+D` (Unix) / `Ctrl+Z` (Windows)

Python 3.x Interactive Shell

```
>>>  2 + 3
5                                          # result printed automatically
>>>  "Gate" + "XAIML"
'GateXAIML'
>>>  _
'GateXAIML'                                # _ = last result
>>>  x = [10, 20, 30]                      # assignment - no output
>>>  help(len)
Help on built-in function len...
>>>  dir(x)
['__add__', ..., 'append', 'clear', ...]
>>>  exit()
```

Example 19: REPL — Underscore and Quick Exploration

```
>>> 100 / 7
14.285714285714286
>>> round(_, 2)          # _ holds last result
14.29
>>> import math
>>> math.factorial(10)
3628800
>>> _ / math.factorial(9)    # _ = 3628800
10.0
>>> type(_)
<class 'float'>
>>> dir(math)[:6]
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']
```

1.1.2.2 Script Mode and `__name__`

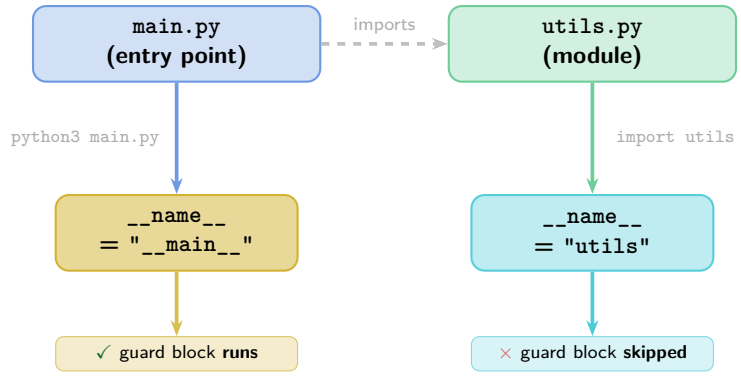
Script Mode and `if __name__ == "__main__"`

Script mode: Python executes a `.py` file top to bottom when run as:

`python3 script.py`
Every module has a built-in attribute `__name__`:

- When a file is **run directly**: `__name__` is set to `"__main__"`
- When a file is **imported**: `__name__` is set to the *module name* (e.g. `"utils"`)

The guard `if __name__ == "__main__":` lets you write code that runs *only when the file is the entry point*, not when it is imported as a library. This is the standard Python idiom for making a file both **importable** and **executable**.



Script Mode Execution Characteristics

- Command-line arguments passed via `sys.argv` (list of strings; `sys.argv[0]` is the script name)
- `python3 -c "print('hello')"` runs a one-liner without a file
- `python3 -m module` runs a module as a script (e.g. `python3 -m http.server`)
- `python3 -i script.py` runs the script then drops into REPL
- Return code: 0 on success; non-zero on error (`sys.exit(code)`)

Example 20: Script with `__main__` Guard

```python
# ---------- utils.py ----------
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

if __name__ == "__main__":
    # Only runs when: python3 utils.py
    # Skipped when:   import utils
    print("Testing utils module...")
    print(add(3, 4))       # 7
    print(multiply(3, 4))  # 12
```

```python
# ---------- main.py ----------
import utils                    # __name__ in utils is "utils"
                                # guard block does NOT run

result = utils.add(10, 20)
print(result)                   # 30
```

Example 21: Command-line Arguments with `sys.argv`

```python
# calculator.py
import sys

if len(sys.argv) != 4:
    print("Usage: python3 calculator.py <num1> <op> <num2>")
    sys.exit(1)

a  = float(sys.argv[1])
op =       sys.argv[2]
b  = float(sys.argv[3])

ops = {'+': a+b, '-': a-b, '*': a*b, '/': a/b}
if op not in ops:
```

```
    print(f"Unknown operator: {op}")
    sys.exit(2)

print(f"{a} {op} {b} = {ops[op]}")

# Usage:  python3 calculator.py 10 + 3.5
# Output: 10.0 + 3.5 = 13.5
```

1.2 Variables & Assignment

1.2.1 Naming Rules

1.2.1.1 Identifier Rules

Python Identifier Rules

An **identifier** is a name given to a variable, function, class, or module. Python's lexical rules:

1. Must begin with a **letter** (`a-z`, `A-Z`) or **underscore** (`_`)
2. Subsequent characters may be **letters, digits (0–9), or underscores**
3. **No spaces, hyphens, or special characters**
4. **Case-sensitive**: `count`, `Count`, `COUNT` are three distinct names
5. **Reserved keywords** cannot be used as identifiers

Keywords (Python 3.12): `False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield`

Valid Identifier	Invalid Identifier	Reason Invalid
`my_var`	`1var`	starts with digit
`_private`	`my-var`	hyphen not allowed
`Counter`	`class`	reserved keyword
`x2`	`my var`	space not allowed
`GATE_DA`	`first$`	special character

Naming Conventions (PEP 8)

- `snake_case` — variables and functions: `total_marks`, `get_data()`
- `PascalCase` (CapWords) — classes: `NeuralNetwork`, `DataLoader`
- `UPPER_SNAKE_CASE` — constants: `MAX_EPOCHS`, `PI`

Example 22: Identifier Validity and `keyword` Module

```
import keyword

# Check all keywords
print(keyword.kwlist)
```

```python
# ['False', 'None', 'True', 'and', 'as', 'assert', ...]

# Check if a name is a keyword
print(keyword.iskeyword("for"))       # True
print(keyword.iskeyword("forEach"))   # False  (valid identifier)
print(keyword.iskeyword("GATE"))      # False

# Python is case-sensitive
Value = 100
value = 200
VALUE = 300
print(Value, value, VALUE)    # 100 200 300 -- three separate variables

# Soft keywords (Python 3.10+): valid identifiers, context-dependent
# match, case, type -- can still be used as variable names
match = 5        # valid (match is a soft keyword)
case  = "test"   # valid
print(match, case)
```

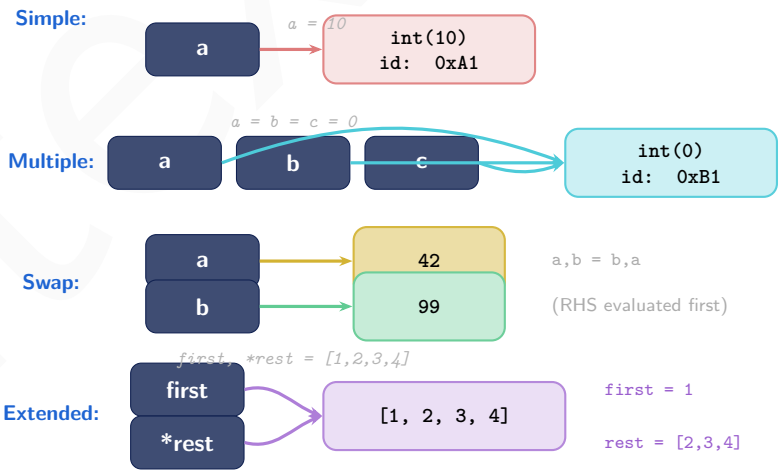
1.2.2 Assignment Mechanics

1.2.2.1 Variable as Name Binding

Variable = Name Bound to an Object

Unlike C/Java where a variable is a *typed memory box*, a Python variable is simply a **name** stored in a *namespace dictionary* that **refers (points) to an object** in memory.
Assignment does not copy data — it binds a name to an existing (or newly created) object.
Consequences:

- Re-assigning a name does not modify the old object
- Multiple names can point to the same object
- Passing a variable to a function passes the *reference* (call-by-object-reference / call-by-sharing)



Assignment Forms in Python

- **Simple**: `x = 10`
- **Multiple (chained)**: `a = b = c = 0` — all names bound to the *same* object
- **Augmented**: `x += 1` — equivalent to `x = x + 1` for immutables; in-place mutation for mutables (list, set, dict)

- **Tuple/simultaneous**: `a, b = 1, 2` — RHS fully evaluated before any binding
- **Extended unpacking**: `first, *rest = iterable` — `*` captures remaining items as a list
- **Walrus operator (:=, PEP 572, Python 3.8+)**: assigns and returns in one expression — used in `while`/`if`

Example 23: All Assignment Forms in Action

```
# Simple
x = 10

# Multiple (all point to the same object)
a = b = c = []
a.append(1)
print(b)            # [1]  -- b IS a (same object!)

# Augmented assignment
n = 5
n += 3              # n = 8
n **= 2             # n = 64
n //= 7             # n = 9
print(n)            # 9

# Tuple / simultaneous swap
x, y = 10, 20
x, y = y, x        # swap -- RHS evaluated first as (20,10)
print(x, y)         # 20 10

# Extended unpacking
first, *middle, last = [1, 2, 3, 4, 5]
print(first)        # 1
print(middle)       # [2, 3, 4]
print(last)         # 5

# Nested unpacking
(a, b), c = (1, 2), 3
print(a, b, c)      # 1 2 3

# Walrus operator (Python 3.8+)
data = [1, 3, 7, 9, 2]
if (n := len(data)) > 3:
    print(f"List has {n} elements")    # List has 5 elements
```

Example 24: Augmented Assignment — Mutable vs Immutable Difference

```
# IMMUTABLE (int, str, tuple): += creates a new object
x = 5
print(id(x))    # e.g., 9789408
x += 1
print(id(x))    # different id -- new int object created

# MUTABLE (list): += mutates IN-PLACE (__iadd__)
lst = [1, 2, 3]
print(id(lst))    # e.g., 140391234
lst += [4, 5]     # calls lst.__iadd__([4,5]) -- same object!
print(id(lst))    # SAME id
print(lst)        # [1, 2, 3, 4, 5]

# Compare with lst = lst + [4,5] which creates a NEW list
lst2 = [1, 2, 3]
old_id = id(lst2)
lst2 = lst2 + [4, 5]    # creates new object
print(id(lst2) == old_id)  # False
```

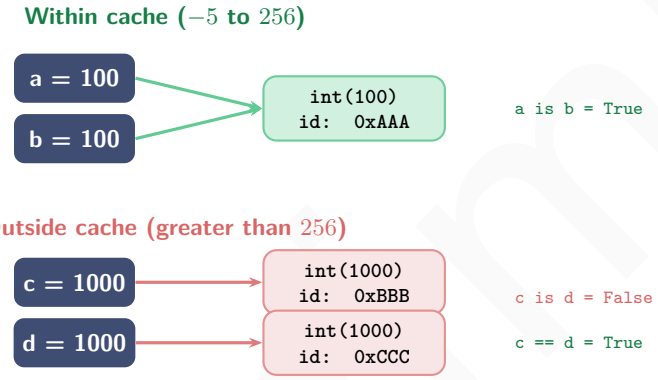
1.2.2.2 Object Identity vs Equality

== vs is — Equality vs Identity

- == calls __eq__: tests whether two objects have the *same value*
- is compares id(): tests whether two names refer to the *exact same object in memory*

Rule of thumb:

- Use == for *value* comparison in almost all cases
- Use is only for None, True, False singletons
- Never use is to compare integers, strings, or custom objects (behaviour depends on implementation internals)



Integer Caching (CPython Implementation Detail)

- CPython pre-allocates **singleton objects** for integers -5 to 256
- Any reference to these values in CPython always points to the *same cached object*
- Outside this range, each literal may create a new object
- **This is an implementation detail of CPython**, not guaranteed by the Python language specification
- **String interning** works similarly: short strings and identifiers are often interned (same object); can be forced with sys.intern()

Example 25: is vs == — When They Differ

```
# Cached integers: is works (coincidentally)
a, b = 100, 100
print(a == b)     # True  (same value)
print(a is b)     # True  (same cached object in CPython)

# Uncached integers: is WRONG to use
c = 1000
d = 1000
print(c == d)     # True   (correct: value comparison)
print(c is d)     # False  (different objects -- NEVER rely on this!)

# None is a singleton -- use 'is'
def maybe():
    pass   # returns None implicitly

result = maybe()
print(result == None)     # True  (works but bad style)
print(result is None)     # True  (correct idiom)
print(result is not None)# False
```

```python

# List comparison
x = [1, 2, 3]
y = [1, 2, 3]
z = x
print(x == y)     # True   (same values)
print(x is y)     # False  (different objects)
print(x is z)     # True   (z IS x -- same object)
```

Example 26: `id()` and Memory Address

```python
import sys

x = "hello"
y = "hello"
# CPython interns short string literals
print(x is y)          # True (interned)
print(id(x), id(y))    # same address

# Force separate objects
a = "hello world"
b = "hello world"
# May or may not be interned -- don't rely on it
print(a is b)          # True in some Python versions (compile-time intern)

# Mutable objects are never shared (different id)
p = [1, 2]
q = [1, 2]
print(p is q)          # False always
print(id(p) != id(q)) # True

# sys.intern forces string interning
import sys
s1 = sys.intern("gate exam result")
s2 = sys.intern("gate exam result")
print(s1 is s2)        # True -- guaranteed by intern
```

1.3 Data Types

1.3.1 Numeric Types

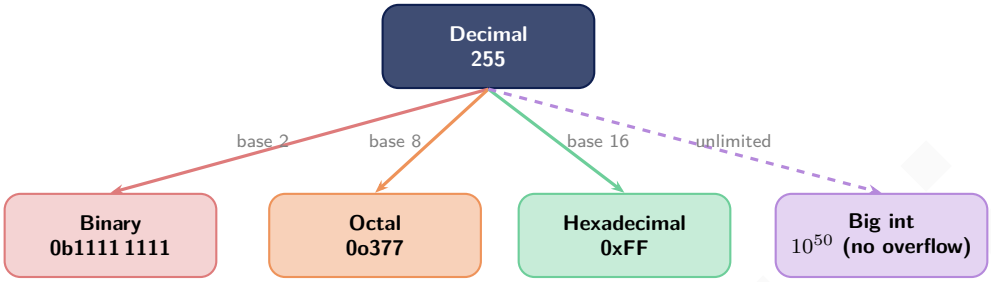
1.3.1.1 `int` — Arbitrary-Precision Integer

`int` — Definition and Precision

Python's `int` type stores **arbitrary-precision integers** — there is *no fixed size, no overflow, no INT_MAX*. The only limit is available memory.
Internally CPython represents small integers as C `long` and switches to a multi-digit "bignum" representation for large values (using base 2^{30} digits).
Literal forms:

Base	Prefix	Example
Decimal	(none)	255
Binary	0b or 0B	0b11111111
Octal	0o or 0O	0o377
Hexadecimal	0x or 0X	0xFF

Underscore separator allowed anywhere for readability: `1_000_000`, `0xFF_AA_BB`.



Key `int` Properties and Methods

- **No overflow**: `2**1000` is exact (a 302-digit number)
- **`bin(n)`, `oct(n)`, `hex(n)`**: convert to string in respective base
- **`int(s, base)`**: parse a string in given base: `int("FF", 16)` → 255
- **`n.bit_length()`**: number of bits needed to represent n
- **`n.bit_count()` (3.10+)**: count of set bits (popcount)
- **`n.to_bytes(length, byteorder)`**: convert to bytes object
- **`divmod(a,b)`**: returns `(a//b, a%b)` in one call
- **Underscore literal**: `1_000_000` for readability (ignored by Python)

Example 27: `int` — Literals, Large Numbers, Bases

```python
# Bases
print(0b1010)          # 10   (binary)
print(0o17)            # 15   (octal)
print(0xFF)            # 255  (hex)
print(0b1111_1111)     # 255  (binary with underscore)

# Arbitrary precision
big = 2 ** 100
print(big)             # 1267650600228229401496703205376
print(type(big))       # <class 'int'>

# No overflow (contrast with C int)
print(10 ** 50)
# 100000000000000000000000000000000000000000000000000

# int methods
n = 255
print(bin(n))               # '0b11111111'
print(oct(n))               # '0o377'
print(hex(n))               # '0xff'
print(n.bit_length())       # 8
print((0b101010).bit_count())  # 3  (three 1-bits)

# Parsing from string
print(int("FF", 16))    # 255
print(int("1010", 2))   # 10
print(int("17", 8))     # 15

# divmod
q, r = divmod(17, 5)
print(q, r)             # 3  2  (17 = 3*5 + 2)
```

1.3.1.2 `float` — IEEE 754 Double Precision

`float` — Definition and IEEE 754

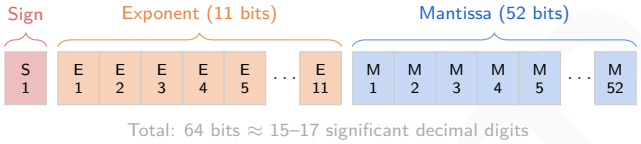
Python's `float` corresponds to a C **double** (64-bit IEEE 754):

- 1 sign bit, 11 exponent bits, 52 mantissa bits
- Range: approximately $\pm 1.8 \times 10^{308}$
- Precision: $\approx$ 15–17 significant decimal digits

Key consequence: many decimal fractions cannot be represented exactly in binary floating point:

$$0.1 + 0.2 = 0.30000000000000004 \neq 0.3$$

This is *not a Python bug* — it is fundamental to IEEE 754.



`float` Properties and Best Practices

- **Special values**: `float('inf')`, `float('-inf')`, `float('nan')`
- **NaN**: `nan != nan` is `True`; check with `math.isnan(x)`
- **Approximate comparison**: use `math.isclose(a, b, rel_tol=1e-9)` never == for floats
- **`round(x, n)`**: rounds to n decimal places (banker's rounding for `.5`: rounds to nearest even)
- **`Decimal` module**: exact decimal arithmetic (financial/scientific use)
- **`fractions.Fraction`**: exact rational arithmetic
- Scientific notation: `1.5e3` $= 1500.0$, `2.5e-4` $= 0.00025$

Example 28: Float Precision and `math.isclose`

```
import math

% Floating-point representation error
print(0.1 + 0.2)              # 0.30000000000000004
print(0.1 + 0.2 == 0.3)       # False  (!!)
print(math.isclose(0.1+0.2, 0.3))  # True  (correct way)

% Special values
inf = float('inf')
nan = float('nan')
print(inf > 10**100)      # True
print(inf + inf)          # inf
print(inf - inf)          # nan
print(nan == nan)         # False  (NaN is never equal to anything)
print(math.isnan(nan))    # True

% Exact decimal arithmetic
from decimal import Decimal
print(Decimal("0.1") + Decimal("0.2"))    # 0.3  (exact!)

% Fraction arithmetic
from fractions import Fraction
print(Fraction(1,3) + Fraction(1,6))      # 1/2  (exact)
```

```
% round() uses banker's rounding
print(round(0.5))   # 0  (rounds to even)
print(round(1.5))   # 2  (rounds to even)
print(round(2.5))   # 2  (rounds to even)
print(round(3.5))   # 4  (rounds to even)
print(round(3.14159, 2))   # 3.14
```

1.3.1.3 `complex` — Complex Numbers

`complex` — Definition

Python has **native complex number** support. A complex literal is written as `a + bj` (using `j` for the imaginary unit i):

$$\texttt{z = 3 + 4j} \quad \Rightarrow \quad z = 3 + 4i$$

Both real and imaginary parts are stored as `float`.

`complex` Properties

- `z.real`: real part (float)
- `z.imag`: imaginary part (float)
- `abs(z)`: modulus $|z| = \sqrt{a^2 + b^2}$
- `z.conjugate()`: returns $a - bj$
- `complex(a, b)`: constructor: `complex(3, 4)` → `(3+4j)`
- All arithmetic operators work: `+`, `-`, `*`, `/`
- `cmath` module for complex-aware `sqrt`, `exp`, `log`

Example 29: Complex Numbers — Arithmetic and `cmath`

```
import cmath, math

z1 = 3 + 4j
z2 = 1 - 2j

print(z1.real, z1.imag)     # 3.0  4.0
print(abs(z1))              # 5.0  (= sqrt(9+16))
print(z1.conjugate())       # (3-4j)

% Arithmetic
print(z1 + z2)    # (4+2j)
print(z1 * z2)    # (11-2j)  = (3*1-4*(-2)) + (3*(-2)+4*1)j
print(z1 / z2)    # (-1+2j)

% cmath: sqrt of negative number
print(cmath.sqrt(-1))        # 1j
print(cmath.sqrt(-4))        # 2j
print(cmath.exp(1j * math.pi))  # (-1+1.2e-16j)       Euler's formula e^(i  )=-1
```

1.3.1.4 `bool` — Boolean Type

`bool` — Definition and Subclass Relationship

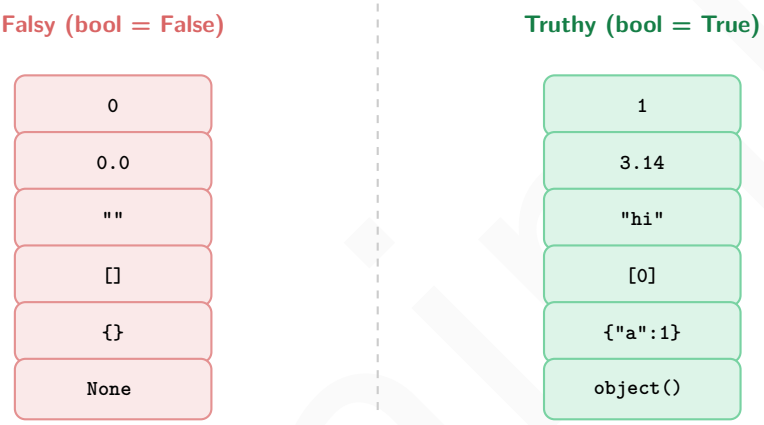
`bool` is a **subclass of** `int`:

$$\texttt{True} \equiv 1, \quad \texttt{False} \equiv 0$$

Only two instances exist: `True` and `False` (singletons).
Truthiness — every object has a boolean interpretation in a conditional:

- **Falsy**: `0, 0.0, 0j, "", [], {}, set(), frozenset(), None, range(0)`
- **Truthy**: everything else (non-zero numbers, non-empty containers, etc.)

Customise via `__bool__` or `__len__` in user-defined classes.



`bool` Properties

- `isinstance(True, int)` returns `True`
- Arithmetic: `True + True == 2`, `False * 100 == 0`
- Useful for counting: loops tracking expressions where variable is greater than 0
- `bool(x)` converts any object to `True`/`False`
- **Always use** `is True` / `is False` only when you need exact boolean singleton check; otherwise use truthiness directly (`if x:` not `if x == True:`)

Example 30: `bool` Arithmetic and Counting

```python
# bool is int subclass
print(type(True))             # <class 'bool'>
print(isinstance(True, int)) # True
print(True + True)            # 2
print(True * 10)              # 10
print(False + 5)              # 5

# Counting with bools
scores = [85, 42, 91, 55, 73, 88, 39]
passing = sum(s >= 60 for s in scores)
print(f"{passing} out of {len(scores)} passed")  # 4 out of 7 passed

# Truthiness in conditions
for val in [0, 1, "", "hi", [], [0], None, 0.0, {}, {"a":1}]:
    print(f"bool({val!r:12}) = {bool(val)}")
# bool(0)           = False
# bool(1)           = True
# bool('')          = False
```

```
# bool('hi')        = True
# ...
```

1.3.2 NoneType

None — The Null Singleton

None is the **sole instance** of NoneType. It represents the *absence of a value*.
Always check with is, never ==:

- x is None — **correct**
- x == None — works but violates PEP 8 (and can be overridden by a class's __eq__)

Functions with **no return statement** implicitly return None.

None Properties

- **Singleton**: id(None) is constant; None is None always True
- **Falsy**: bool(None) == False
- Common uses: default function argument, sentinel value, uninitialised state
- **Not the same** as 0, False, "", [] (though all are falsy)

Example 31: None — Implicit Return and Sentinel

```
# Implicit None return
def no_return():
    x = 10    # does something but no return

result = no_return()
print(result)            # None
print(type(result))      # <class 'NoneType'>

# None as default / sentinel
def search(lst, target):
    for i, val in enumerate(lst):
        if val == target:
            return i
    return None    # explicit None when not found

idx = search([10, 20, 30], 20)
if idx is not None:
    print(f"Found at index {idx}")    # Found at index 1

# None as optional argument default
def greet(name, title=None):
    if title is None:
        return f"Hello, {name}!"
    return f"Hello, {title} {name}!"

print(greet("Alice"))              # Hello, Alice!
print(greet("Alice", "Dr."))       # Hello, Dr. Alice!
```

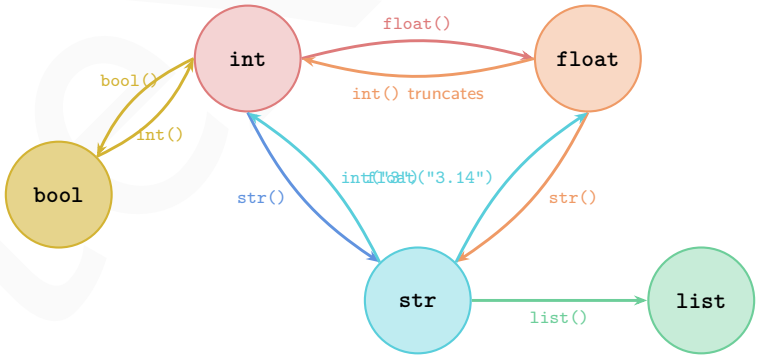
1.3.3 Type Conversion

Implicit vs Explicit Type Conversion

- **Implicit (coercion)**: Python *automatically* converts when safe and unambiguous (only a few cases: integer added to float yields a float; bool in arithmetic)
- **Explicit (casting)**: programmer manually calls a constructor function to convert types

Key explicit conversion functions:

Function	Converts to	Notes
`int(x)`	`int`	truncates float; parses str (base 10)
`float(x)`	`float`	parses str; `float("inf")` valid
`str(x)`	`str`	calls `__str__`
`bool(x)`	`bool`	falsy/truthy rules
`list(x)`	`list`	any iterable
`tuple(x)`	`tuple`	any iterable
`set(x)`	`set`	any iterable; removes duplicates
`dict(x)`	`dict`	iterable of key-value pairs
`bytes(x)`	`bytes`	int (length), iterable of ints, or str+encoding
`chr(n)`	`str`	Unicode code point to char
`ord(c)`	`int`	char to Unicode code point



Type Inspection Functions

- `type(x)`: returns the exact class; does *not* consider inheritance (`type(True) is bool`, not `int`)
- `isinstance(x, T)`: returns `True` if `x` is an instance of `T` *or any subclass* of `T`; accepts a tuple of types: `isinstance(x, (int, float))`
- `issubclass(A, B)`: returns `True` if class `A` is a subclass of `B`; `issubclass(bool, int)` is `True`
- **Prefer** `isinstance` over `type(x) ==` for type-checking (more Pythonic; respects inheritance)

Example 32: Explicit Conversion — Common Cases

```python
# int conversions
print(int(3.9))        # 3   (truncates toward zero, NOT floor)
print(int(-3.9))       # -3  (truncates toward zero)
print(int("42"))       # 42
print(int("0xFF", 16))# 255
print(int(True))       # 1

# float conversions
print(float("3.14"))  # 3.14
print(float("1e-5"))  # 1e-05
print(float(True))    # 1.0

# str conversions
print(str(100))        # '100'
print(str([1,2,3]))    # '[1, 2, 3]'
print(str(None))       # 'None'

# bool conversions
print(bool(0))         # False
print(bool(-1))        # True
print(bool(""))        # False
print(bool("False"))   # True  (non-empty string!)

# list / set / tuple
print(list("gate"))    # ['g', 'a', 't', 'e']
print(set([1,2,2,3])) # {1, 2, 3}
print(tuple(range(5)))# (0, 1, 2, 3, 4)

# chr and ord
print(ord('A'))        # 65
print(chr(65))         # 'A'
print(chr(ord('a')+5))# 'f'
```

Example 33: `type()` vs `isinstance()`

```python
print(type(True) == int)         # False  (bool, not int)
print(type(True) == bool)        # True
print(isinstance(True, int))     # True   (bool is subclass of int)
print(isinstance(True, bool))    # True

# isinstance with tuple of types
def process(x):
    if isinstance(x, (int, float)):
        return x ** 2
    elif isinstance(x, str):
        return x.upper()
    else:
        raise TypeError(f"Unsupported: {type(x)}")

print(process(5))        # 25
print(process(3.14))     # 9.8596
print(process("gate"))   # GATE

% issubclass
print(issubclass(bool, int))      # True
print(issubclass(int, object))    # True
print(issubclass(float, int))     # False
```

1.4 Operators

1.4.1 Arithmetic Operators

1.4.1.1 All Arithmetic Operators

Python Arithmetic Operators

Op	Name	Example	Result
+	Addition	7 + 3	10
-	Subtraction	7 - 3	4
*	Multiplication	7 * 3	21
/	True Division	7 / 3	2.333... (always float)
//	Floor Division	7 // 3	2 (towards $-\infty$)
%	Modulo	7 % 3	1 (sign follows divisor)
**	Exponentiation	2 ** 10	1024

Critical points:

- / *always* returns `float`: `6/2 = 3.0`
- // floors *towards* negative infinity: `-7 // 2 = -4` (not -3)
- % sign matches *divisor*: `-7 % 2 = 1` (not -1)
- **Invariant**: `(a // b) * b + (a % b) == a` always holds
- ** is right-associative: `2**3**2 = 2**(3**2) = 512`

Floor Division and Modulo Sign Rules

- **Floor division** rounds towards negative infinity (not towards zero):
  `7 // 2 = 3`, `-7 // 2 = -4`, `7 // -2 = -4`, `-7 // -2 = 3`
- **Modulo** sign matches the *divisor* (not dividend):
  `7 % 2 = 1`, `-7 % 2 = 1`, `7 % -2 = -1`, `-7 % -2 = -3`
- **Invariant check**: `(a // b) * b + a % b == a`
- ** **is right-associative**: `a**b**c = a**(b**c)`

Example 34: Arithmetic — Tricky Floor Division and Modulo

```
# True division always returns float
print(7 / 2)      # 3.5
print(6 / 2)      # 3.0  <-- float, not int!
print(6 / 3)      # 2.0

# Floor division towards -inf
print( 7 // 2)    #  3
```

```python
print(-7 // 2)    # -4  (not -3!)
print( 7 //-2)    # -4
print(-7 //-2)    #  3

# Modulo sign follows divisor
print( 7 % 2)     #  1
print(-7 % 2)     #  1  (not -1!)
print( 7 %-2)     # -1
print(-7 %-2)     # -3

# Invariant: always holds
for a, b in [(7,2),(-7,2),(7,-2),(-7,-2)]:
    assert (a//b)*b + a%b == a, "invariant broken!"
print("Invariant holds for all cases")

# ** right-associative
print(2 ** 3 ** 2)    # 512  = 2**(3**2) = 2**9
print((2**3) ** 2)    # 64   = 8**2

# Exponentiation with float
print(16 ** 0.5)      # 4.0  (square root)
print(8 ** (1/3))     # 2.0  (cube root)
```

1.4.2 Comparison Operators

1.4.2.1 Comparison and Chaining

Comparison Operators

All comparison operators return `True` or `False`:

Op	Meaning	Example (True)
==	Equal to	3 == 3
!=	Not equal to	3 != 4
<	Less than	2 < 5
>	Greater than	5 > 2
<=	Less or equal	3 <= 3
>=	Greater or equal	4 >= 3

Chaining: Python allows chaining: `1 < x < 10` is evaluated as `(1 < x) and (x < 10)` — both conditions must hold. No redundant evaluation.

Example 35: Comparison Chaining

```python
x = 5
# Chained comparison
print(1 < x < 10)        # True   == (1<5) and (5<10)
print(1 < x < 5)         # False  == (1<5) and (5<5) = True and False
print(1 < 2 < 3 < 4)     # True
print(10 > x >= 5)       # True

# Useful for range checks
def classify_score(score):
    if 90 <= score <= 100:
        return "A"
    elif 70 <= score < 90:
```

```
        return "B"
    elif 50 <= score < 70:
        return "C"
    else:
        return "F"

print(classify_score(85))   # B
print(classify_score(50))   # C
print(classify_score(100))  # A
```

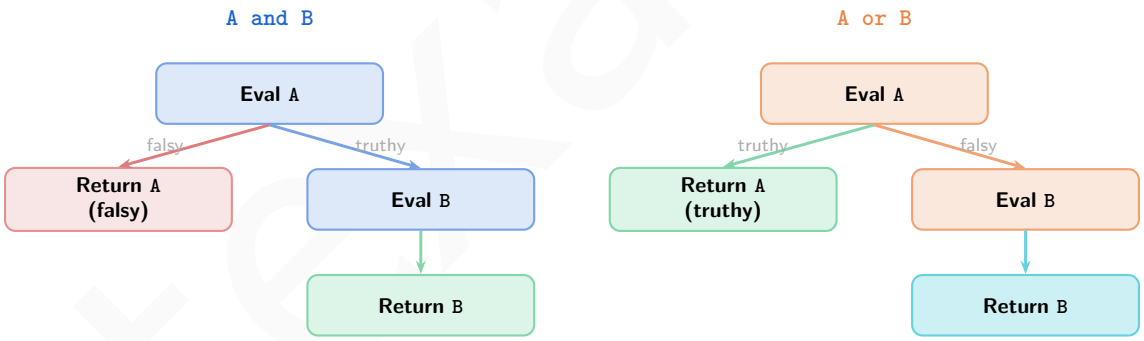
1.4.3 Logical Operators

1.4.3.1 Short-Circuit Evaluation

Logical Operators and Short-Circuit Evaluation

Op	Behaviour	Returns
and	`False` ⇒ stop; else continue	last evaluated operand
or	`True` ⇒ stop; else continue	last evaluated operand
not	inverts truthiness	`True` or `False`

Critical: `and` and `or` return the *last evaluated 1386operand*, not necessarily a boolean. This enables idiomatic patterns.
Short-circuit prevents evaluation of later expressions if result is already determined.



Logical Operator Patterns (Idiomatic Python)

- **Default value**: `x = val or default` — use `default` if `val` is falsy
- **Conditional call**: `obj and obj.method()` — only call if `obj` is not `None`/falsy
- **Ternary equivalent**: `a if cond else b` (preferred over `cond and a or b` which has a bug when `a` is falsy)
- **Guard check**: `lst and lst[0]` — safe first-element access
- **All-falsy**: `not any(lst)` is equivalent to `all(not x for x in lst)`

Example 36: Short-Circuit Evaluation — Return Value and Side Effects

```
# and returns first falsy or last value
print(0 and 5)        # 0     (0 is falsy -- short-circuit)
print(3 and 5)        # 5     (3 truthy; return last: 5)
```

```python
print(3 and 0)          # 0     (3 truthy; return last: 0)
print("" and "hi")      # ''    (empty string falsy)

# or returns first truthy or last value
print(0 or 5)           # 5     (0 falsy; try next)
print(3 or 5)           # 3     (3 truthy -- short-circuit)
print(0 or "")          # ''    (both falsy; return last)
print([] or {})         # {}    (both falsy; return last)

# Practical patterns
name = ""
display = name or "Anonymous"
print(display)    # Anonymous

config = None
value = config and config.get("key")
print(value)      # None  (config is None, short-circuit)

# Short-circuit prevents errors
lst = []
first = lst and lst[0]    # safe: lst is falsy, lst[0] never evaluated
print(first)              # []

# NOT always returns bool
print(not 0)      # True
print(not 5)      # False
print(not "")     # True
print(not [1])    # False
```

1.4.4 Bitwise Operators

1.4.4.1 Bit-Level Operations

Bitwise Operators

Op	Name	Example	Result
&	AND	0b1010 & 0b1100	0b1000 = 8
|	OR	0b1010 | 0b1100	0b1110 = 14
^	XOR	0b1010 ^ 0b1100	0b0110 = 6
~	NOT (complement)	~5	−6 (two's complement)
«	Left shift	3 « 2	12 (3×2^2)
»	Right shift	12 » 2	3 $(12 \div 2^2)$

`~n = -(n+1)` (bitwise NOT uses two's complement)

A	B	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Example: 10 & 12

1	0	=10	(01010)
1	1	=12	(01100)
1	0	=8	(01000)

Example 37: Bitwise Operators — Practical Uses

```python
a, b = 0b1010, 0b1100   # 10 and 12

print(bin(a & b))     # 0b1000  = 8   (AND: bits set in both)
print(bin(a | b))     # 0b1110  = 14  (OR:  bits set in either)
print(bin(a ^ b))     # 0b0110  = 6   (XOR: bits set in exactly one)
print(bin(~a))        # -0b1011 = -11 (NOT: ~n = -(n+1))

# Shifts = fast multiply/divide by powers of 2
print(3 << 2)         # 12   = 3 * 4
print(12 >> 2)        # 3    = 12 // 4
print(1 << 8)         # 256  = 2^8

# Practical: check if a number is even (fast)
n = 18
print(n & 1)          # 0 --> even (last bit is 0)
n = 17
print(n & 1)          # 1 --> odd

# Bit masking -- extract lower 4 bits
value = 0b10110111
lower4 = value & 0b00001111
print(bin(lower4))    # 0b111  = 7

# Swap without temp using XOR
x, y = 5, 9
x ^= y;  y ^= x;  x ^= y
print(x, y)           # 9 5
```

1.4.5 Identity & Membership Operators

1.4.5.1 `is`, `in` and their Negations

Identity and Membership Operators

Op	Tests	Calls internally
`is`	Same object (`id(a)==id(b)`)	— (identity check)
`is not`	Different objects	—
`in`	Membership in container	`__contains__`
`not in`	Non-membership	`__contains__`

Complexity of `in`:

Container	Average complexity
`list`, `tuple`	$O(n)$ — linear scan
`set`, `dict`	$O(1)$ — hash-based
`str`	$O(n)$ — substring search

Example 38: Identity and Membership — All Containers

```python
# Membership in various containers
print(3 in [1, 2, 3, 4])          # True  (list)
```

```python
print(5 not in (1, 2, 3))          # True  (tuple)
print("gate" in {"gate", "exam"}) # True  (set)
print("name" in {"name": "Alice"})# True  (dict keys)
print("lo" in "hello")             # True  (substring)

# Performance: set lookup O(1) vs list O(n)
import time
data_list = list(range(10_000_000))
data_set  = set(range(10_000_000))

t0 = time.perf_counter()
_ = 9_999_999 in data_list
print(f"list: {time.perf_counter()-t0:.4f}s")   # ~0.1s

t0 = time.perf_counter()
_ = 9_999_999 in data_set
print(f"set : {time.perf_counter()-t0:.6f}s")   # ~0.000001s

# is: identity
a = [1, 2, 3]
b = a           # same object
c = [1, 2, 3]  # equal but different object

print(a is b)   # True
print(a is c)   # False
print(a == c)   # True
```

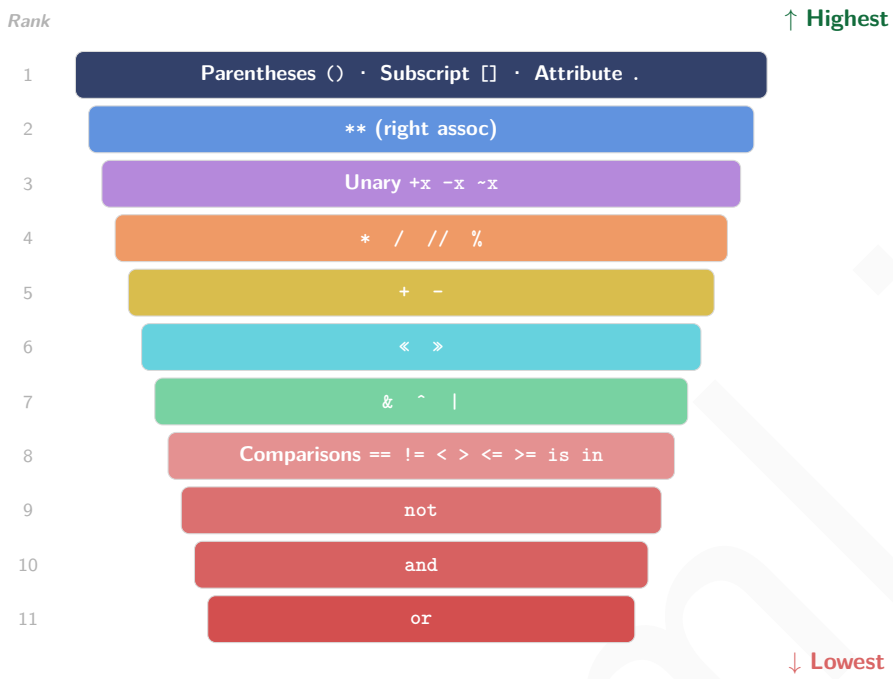
1.4.6 Operator Precedence

1.4.6.1 Full Precedence Table

Python Operator Precedence (Highest to Lowest)

Level	Operators	Associativity
1 (highest)	`() [] .` (call, subscript, attr)	left
2	`**`	**right**
3	`+x -x ~x` (unary)	right
4	`* / // %`	left
5	`+ -`	left
6	`« »`	left
7	`&`	left
8	`^`	left
9	`|`	left
10	`== != < > <= >= is in not in is not`	left
11	`not`	right
12	`and`	left
13 (lowest)	`or`	left

Walrus `:=` has lowest priority after `or` in most contexts



Example 39: Precedence — Surprising Evaluations

```python
# ** before unary minus
print(-2 ** 2)      # -4   = -(2**2), NOT (-2)**2 = 4
print((-2) ** 2)    # 4    -- explicit override

# Comparison before logical
print(2 + 3 > 4 and 10 - 1 < 15)
# = (5 > 4) and (9 < 15) = True and True = True

# not before and before or
print(True or False and False)
# = True or (False and False) = True or False = True
print(not False or True and False)
# = (not False) or (True and False)
# = True or False = True

# Tricky: chained comparison
print(1 < 2 == 2 < 3)    # True (chained: 1<2 and 2==2 and 2<3)

# is before not
x = None
print(x is not None)    # False  (is not is ONE operator, not "is" and "not")
print(not x is None)    # False  but means not (x is None) -- confusing!
# Always use: x is not None  (not: not x is None)

# Bitwise before comparison
print(2 & 3 == 3)     # True: 2 & (3 == 3) = 2 & True = 2 & 1 = 0? No!
# Actually: (2 & 3) == 3 ... wait: precedence: == is LOWER than &
# 2 & 3 = 2; 2 == 3 = False  -- correct parse
print((2 & 3) == 3)  # False  (2 & 3 = 2, not 3)
print(2 & (3 == 3))  # 2 & True = 2 & 1 = 0... as int, but this is confusing
# Use parentheses whenever mixing bitwise and comparison
```

Example 40: Precedence — Using Parentheses Correctly

```python
# Misleading without parens
result = 5 + 3 * 2 ** 2 - 1
# = 5 + 3 * 4 - 1
# = 5 + 12 - 1
```

```
# = 16
print(result)   # 16

# With explicit parens for clarity
result = 5 + (3 * (2 ** 2)) - 1
print(result)   # 16 (same)

# Different meaning with different parens
print((5 + 3) * (2 ** 2) - 1)    # 8 * 4 - 1 = 31
print(5 + 3 * 2 ** (2 - 1))      # 5 + 3*2 = 11

# Short-circuit with mixed operators
x = 5
print(x > 3 and x < 10 or x == 5)
# = (x>3 and x<10) or (x==5)
# = (True and True) or True
# = True or True = True

# Rule: when in doubt, use parentheses!
```

1.5 Problems

Problem 1 ***[MCQ]*** *What is the output of the following code?*

```
x = 257
y = 257
print(x is y)
```

(A) `True`*, because* `x` *and* `y` *hold equal values*

(B) `False`*, because integers outside* $[-5, 256]$ *are not cached by CPython*

(C) `True`*, because Python interns all integer literals*

(D) `True`*, because* `is` *compares values for immutable types*

Problem 2 ***[MCQ]*** *Which statement correctly describes a Python variable?*

(A) A variable is a fixed memory location that stores a value.

(B) A variable is a name bound *to an object; the object lives on the heap.*

(C) A variable is typed at declaration and cannot be rebound to a different type.

(D) A variable is a pointer that is dereferenced automatically.

Problem 3 ***[MCQ]*** *After executing the following code, what are the values of* `a` *and* `b`*?*

```
a = 5
b = 10
a, b = b, a
```

(A) `a = 5, b = 10`

(B) `a = 10, b = 5`

(C) `a = 10, b = 10`

(D) Runtime error — simultaneous assignment is not allowed

Problem 4 ***[MCQ]*** *What does the following extended unpacking produce?*

```
first, *rest = [10, 20, 30, 40]
print(first, rest)
```

(A) `10 [20, 30, 40]`

(B) `10 (20, 30, 40)`

(C) `[10] [20, 30, 40]`

(D) `10 20`

Problem 5 ***[MSQ]*** *Which of the following are* ***valid*** *Python identifiers?*

(A) `_myVar`

(B) `2fast`

(C) `camelCase99`

(D) `class`

(E) `MAX_SIZE`

Problem 6 ***[MCQ]*** *What is the output of the following snippet?*

```
a = b = c = 0
a += 1
print(a, b, c)
```

(A) `1 1 1`

(B) `1 0 0`

(C) `0 0 1`

(D) Runtime error

Problem 7 ***[MCQ]*** *Consider:*

```
x = 10
y = 10
print(x is y, x == y)
```

What is printed? (Assume standard CPython behaviour.)

(A) `True True`

(B) `False True`

(C) `True False`

(D) `False False`

Problem 8 ***[NAT]*** *How many objects are present after executing the following lines?*

```
p = 3\\
q = p\\
r = 3\\
s = 4\\
```

(Count distinct objects.)

Problem 9 ***[MCQ]*** *What is the result of `-13 // 2` in Python?*

(A) `-7`

(B) `-6`

(C) `-6.5`

(D) `6.5`

Problem 10 ***[MCQ]*** *What is the value of `-7 % 2` in Python?*

(A) `-1`

(B) `1`

(C) `-3`

(D) `3`

Problem 11 ***[MCQ]*** *Which of the following numeric literals is* ***not*** *a valid Python `int` literal?*

(A) `0b1010`

(B) `0o17`

(C) `0xFF`

(D) `0d42`

Problem 12 ***[MCQ]*** *What is the output of the following code?*

```
print(True + True + False)
```

(A) `True`

(B) `1`

(C) `2`

(D) `3`

Problem 13 ***[MCQ]*** *Which statement about Python `float` is* ***correct****?*

(A) `float` *uses arbitrary precision, like* `int`*.*

(B) `0.1 + 0.2 == 0.3` *evaluates to* `True` *in CPython.*

(C) `float` *follows IEEE 754 double-precision standard.*

(D) `float('nan') == float('nan')` *evaluates to* `True`*.*

Problem 14 ***[MCQ]*** *What is the output?*

```
z = 3 + 4j
print(abs(z))
```

(A) `7`

(B) `7j`

(C) `5.0`

(D) `25.0`

Problem 15 ***[MSQ]*** *Which of the following are* ***falsy*** *in Python?*

(A) `0`

(B) `0.0`

(C) `"False"`

(D) `None`

Problem 16 ***[MCQ]*** *What is the correct way to check whether a variable* `x` *holds* `None`*?*

(A) `x == None`

(B) `x is None`

(C) `type(x) == NoneType`

(D) `not x`

Problem 17 ***[MCQ]*** *Which built-in function returns the* ***memory address*** *of an object?*

(A) `addr()`

(B) `ref()`

(C) `id()`

(D) `ptr()`

Problem 18 ***[MCQ]*** *What is the output of the following?*

```
print(isinstance(True, int))
```

(A) `False`*, because* `bool` *is not a subclass of* `int`

(B) `True`*, because* `bool` *is a subclass of* `int`

(C) `False`*, because* `True` *is a keyword, not an object*

(D) Runtime error

Problem 19 ***[MCQ]*** *What does* `int("0xFF", 16)` *return?*

(A) `255`

(B) `256`

(C) `0`

(D) `ValueError`

Problem 20 ***[MCQ]*** *What is the output of the following?*

```
print(int(3.9))
```

(A) `4` *(rounds to nearest)*

(B) `3` *(truncates toward zero)*

(C) `3.9`

(D) `ValueError`

Problem 21 ***[MCQ]*** *What is the result of* `bool("")`*?*

(A) `True`

(B) `False`

(C) `None`

(D) `ValueError`

Problem 22 ***[MSQ]*** *Which of the following conversions will* ***raise a*** `ValueError`*?*

(A) `int("abc")`

(B) `float("3.14")`

(C) `int("3.7")`

(D) `int("10", 2)`

(E) `float("nan")`

Problem 23 ***[MCQ]*** *What does* `type(1 + 2.0)` *return?*

(A) `<class 'int'>`

(B) `<class 'float'>`

(C) `<class 'complex'>`

(D) `<class 'number'>`

Problem 24 ***[NAT]*** *What is the value of* `2 ** 3 ** 2` *in Python? (Enter an integer.)*

Problem 25 ***[MCQ]*** *What is the output of the following?*

```
print(10 / 3)
```

(A) `3`

(B) `3.0`

(C) `3.3333333333333335`

(D) `3.3333333333333333`

Problem 26 ***[MCQ]*** *What is the result of* `10 // 3`*?*

(A) `3.0`

(B) `3`

(C) `4`

(D) `3.33`

Problem 27 ***[MCQ]*** *Which operator has the* ***highest*** *precedence among the following?*

(A) `+`

(B) `*`

(C) `**`

(D) `%`

Problem 28 ***[NAT]*** *What is the output of `17 % -5`?*

Problem 29 ***[MCQ]*** *What is the value of the expression `3 + 4 * 2 ** 2 - 1`?*

(A) `18`

(B) `28`

(C) `18`

(D) `48`

Problem 30 ***[MCQ]*** *What is the output of the following?*

```
x = 5
print(1 < x < 10)
```

(A) `True`

(B) `False`

(C) `1`

(D) `TypeError`

Problem 31 ***[MCQ]*** *What does the following expression evaluate to?*

```
print(0 or "hello" or "world")
```

(A) `True`

(B) `0`

(C) `hello`

(D) `world`

Problem 32 ***[MCQ]*** *What is the output of the following?*

```
print(5 and 0 and 10)
```

(A) `0`

(B) `5`

(C) `10`

(D) `False`

Problem 33 ***[MSQ]*** *Which of the following expressions evaluate to* ***True****?*

(A) `not 0`

(B) `not ""`

(C) `not None`

(D) `not [1, 2]`

Problem 34 ***[MCQ]*** *What is the output?*

```
x = None
print(x == False)
```

(A) `True`

(B) `False`

(C) `None`

(D) `TypeError`

Problem 35 ***[MCQ]*** *What is the result of comparing `"abc" == "abc"` and `"abc" is "abc"` in the interactive REPL (standard CPython)?*

(A) *Both are `True` due to string interning of literals.*

(B) *Both are `False`.*

(C) *`==` is `True`; `is` is `False`.*

(D) *`==` is `False`; `is` is `True`.*

Problem 36 ***[MCQ]*** *What is the output of the following?*

```
print(~5)
```

(A) `4`

(B) `-4`

(C) `-6`

(D) `6`

Problem 37 ***[NAT]*** *What is the value of `3 << 2`?*

Problem 38 ***[NAT]*** *What is the value of `15 >> 2`?*

Problem 39 ***[MCQ]*** *What is the result of the expression `not 3 > 2`?*

(A) `True`

(B) `False`

(C) `1`

(D) `TypeError`

Problem 40 ***[MCQ]*** *Evaluate: `2 | 3 ^ 1 & 5`*

(A) `2`

(B) `3`

(C) `6`

(D) `7`

Problem 41 ***[MCQ]*** *What is the output of the following code?*

```
print(2 + 3 == 5 and 10 / 2 == 5.0)
```

(A) `True`

(B) `False`

(C) `1`

(D) `5.0`

Problem 42 ***[MCQ]*** *What is the output?*

```
x = 5
x **= 2
x //= 3
print(x)
```

(A) `8`

(B) `8.33`

(C) `8.0`

(D) `9`

Problem 43 ***[MCQ]*** *What is the output?*

```
print(0.1 + 0.1 == 0.2)
```

(A) `True`

(B) `False`

(C) `None`

(D) `TypeError`

Problem 44 ***[MCQ]*** *What function should be used for* ***approximate floating-point comparison****?*

(A) `float.approx()`

(B) `math.isclose()`

(C) `round(a) == round(b)`

(D) `cmp(a, b)`

Problem 45 ***[NAT]*** *What is the value printed by the following snippet?*

```
a, *b, c = [1, 2, 3, 4, 5]
print(len(b))
```

Problem 46 ***[MSQ]*** *Which of the following are* ***augmented assignment operators*** *in Python?*

(A) `+=`

(B) `**=`

(C) `:=`

(D) `«=`

Problem 47 ***[MCQ]*** *What is printed by the following code?*

```
print(type(1_000_000))
```

(A) `<class 'float'>`

(B) `SyntaxError`

(C) `<class 'int'>`

(D) `<class 'str'>`

Problem 48 ***[MCQ]*** *Which of the following correctly describes short-circuit evaluation of* `and`*?*

(A) `and` *always evaluates both operands and returns* `True` *or* `False`*.*

(B) `and` *returns the first operand if it is falsy; otherwise returns the second operand.*

(C) `and` *returns the first operand if it is truthy; otherwise returns the second operand.*

(D) `and` *always returns a boolean.*

Problem 49 ***[MCQ]*** *Which built-in function checks whether an object is an* ***instance of a given class or a subclass thereof****?*

(A) `type()`

(B) `isinstance()`

(C) `issubclass()`

(D) `hasattr()`

Problem 50 ***[MCQ]*** *What is the output of the following?*

```
z = 2 + 3j
print(z.real, z.imag)
```

(A) `2 3`

(B) `2.0 3.0`

(C) `(2, 3)`

(D) `2+3j`

Problem 51 ***[MSQ]*** *Which of the following statements about* `None` *in Python are* ***correct****?*

(A) `None` *is a singleton object.*

(B) `None == False` *evaluates to* `True`*.*

(C) A function with no `return` *statement implicitly returns* `None`*.*

(D) `bool(None)` *is* `False`*.*

Problem 52 ***[NAT]*** *What is the output of the following expression?*

```
print((-7) // 2 + (-7) % 2)
```

(Enter the integer value.)

1.6 YouTube Links and QR Codes

Lecture	Details	YouTube Link	QR Code
01	Introduction to Python	https://youtu.be/0oymy49iVCE	
02	Variables, Identifiers & Assignment	https://youtu.be/lI1WqIlG9sw	
03	Data Types in Python	https://youtu.be/xYy67SiSPXY	
04	Operators in Python	https://youtu.be/nqXLivL24pc	
05	Problem Solving on Variables, Assignment Mechanism & Operators	https://youtu.be/xRmlSKYti8Y	

Chapter 2

Built-in Data Structures

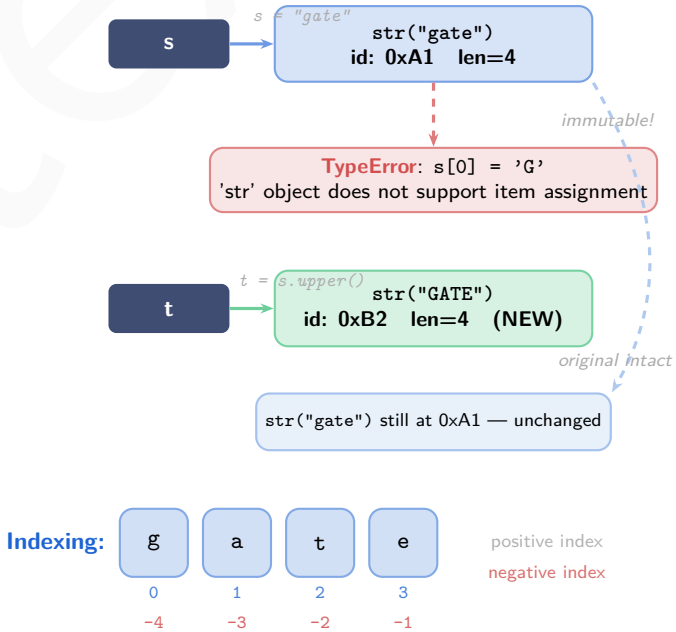
2.1 Strings

2.1.1 String Properties

2.1.1.1 Immutable Unicode Sequence

String — Definition and Core Properties

A Python **string** (`str`) is an **immutable, ordered sequence of Unicode code points**. Every character is a Unicode scalar value (Python 3 is fully Unicode-aware; there is no separate "byte string" in `str`).
Immutable means: once created, the contents of a string *cannot be changed in place*. Any operation that appears to modify a string (replace, upper, strip, ...) actually **returns a brand-new string object**; the original is unchanged.
Sequence means: strings support all sequence operations — indexing, slicing, `len()`, `in`, iteration.



String Properties Summary

- **Immutable**: `s[0] = 'X'` raises `TypeError`; use `s[:0] + 'X' + s[1:]` to build modified string
- **Ordered sequence**: maintains insertion order; supports `len()`, indexing, slicing, iteration
- **Unicode**: each character is a Unicode code point (`ord('A')`=65, `ord('α')`=945)
- **Positive indexing**: `s[0]` first, `s[len(s)-1]` last
- **Negative indexing**: `s[-1]` last, `s[-2]` second-last, `s[-n] = s[len(s)-n]`
- **Hashable**: strings can be dict keys and set members (because they are immutable)
- **Interned**: short strings and identifiers are often cached as a single object in CPython

Example 41: Immutability and Unicode Indexing

```
s = "GateXAIML"
print(len(s))         # 9
print(s[0])           # G
print(s[-1])          # L
print(s[-3])          # I   = s[9-3] = s[6]

# Every char is a full Unicode code point
emoji = "Hi       "
print(len(emoji))     # 4 (each character, including emoji, counts as 1)
print(emoji[-1])      #

# Immutability
try:
    s[0] = 'g'
except TypeError as e:
    print(e)          # 'str' object does not support item assignment

# "Modify" by building new string
s_new = 'g' + s[1:]
print(s_new)          # gateXAIML  (new object)
print(s)              # GateXAIML  (original unchanged)

# Concatenation creates new object each time (use join for many strings)
a = "Hello"
b = " World"
c = a + b             # new str object "Hello World"
print(id(a) == id(c))  # False
```

2.1.2 String Literals

2.1.2.1 Quoting Styles and Escape Sequences

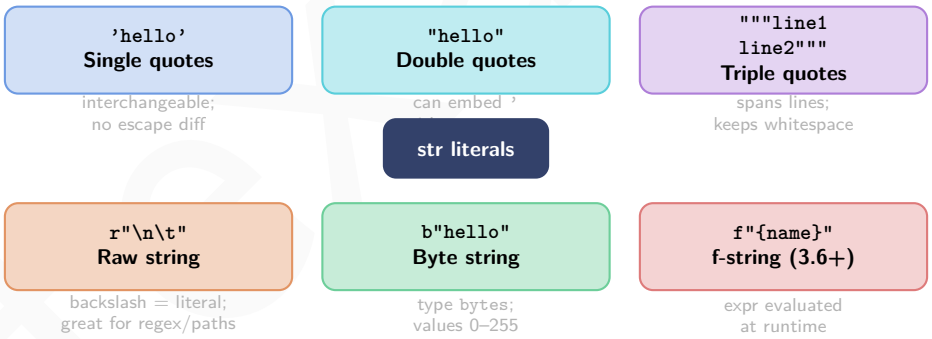
String Literal Forms

Python provides multiple ways to write string literals:

Form	Syntax	Use Case
Single-quote	`'hello'`	Basic strings
Double-quote	`"hello"`	Contains apostrophe: `"it's"`
Triple single	`"'...'"`	Multi-line strings, docstrings
Triple double	`"""..."""`	Multi-line strings, docstrings
Raw string	`r"...\n..."`	Regex, file paths (no escape)
Byte string	`b"hello"`	Binary data, network protocols
f-string	`f"{expr}"`	Formatted output (Python 3.6+)
Raw f-string	`rf"..."`	Combine raw + format

Common escape sequences:

`\n`	newline	`\t`	tab
`\r`	carriage return	`\\`	literal backslash
`\'`	single quote	`\"`	double quote
`\0`	null char	`\uXXXX`	Unicode (4 hex)
`\UXXXXXXXX`	Unicode (8 hex)	`\xHH`	hex byte



Raw Strings and f-Strings Properties

- **Raw string** `r"..."`: backslash is treated as a literal character — `r"\n"` is two chars (\ and `n`), not a newline. Ideal for: regex patterns, Windows file paths
- **Byte string** `b"..."`: type is `bytes`, not `str`; each element is an integer 0–255. Use for binary protocols, file I/O
- **f-string** `f"..."`: *expressions* inside `{...}` are evaluated at runtime; supports full format spec inside the braces: `f"{x:.2f}"`, `f"{name!r}"`
- **f-string = specifier** (Python 3.8+): `f"{x=}"` prints `x=42` for debugging
- Prefix combinations: `rb"..."` (raw bytes), `rf"..."` (raw f-string), `u"..."` (Unicode, legacy Python 2)
- **Adjacent string literals** are *concatenated at compile time*: `"hello" " " "world"` → `"hello world"`

Example 42: Single, Double, Triple Quotes and Escape Sequences

```
# Single and double -- interchangeable
s1 = 'He said "hello"'        # embed double quotes in single
s2 = "it's a test"            # embed apostrophe in double

# Triple quotes -- multi-line
poem = """Roses are red,
Violets are blue,
Python is great,
And so are you!"""
print(poem)

# Triple for long strings with quotes inside
sql = '''SELECT *
FROM users
WHERE name = 'Alice' AND age > 18'''

% Using math mode escape wrapper if Greek character causes engine failures
```

```
# Escape sequences
print("Tab:\there")           # Tab:    here
print("Newline:\nSecond")     # Newline: / Second
print("Backslash: \\")        # Backslash: \
print("Unicode: \u03B1")      # Unicode: α  (Greek alpha)
print("Hex: \x41\x42")        # Hex: AB
```

Example 43: Raw Strings — Regex and File Paths

```
import re, os

# Without raw string: double backslash needed
path1 = "C:\\Users\\Alice\\Documents"
print(path1)   # C:\Users\Alice\Documents

# With raw string: cleaner
path2 = r"C:\Users\Alice\Documents"
print(path2)   # C:\Users\Alice\Documents  (same result)

# Regex: raw strings are essential
email_pattern_bad  = "\\b[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}\\b"
email_pattern_good = r"\b[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}\b"

text = "Contact us at gate@aiml.edu or admin@python.org"
emails = re.findall(email_pattern_good, text)
print(emails)   # ['gate@aiml.edu', 'admin@python.org']

# Raw string still processes some escapes (e.g. cannot end with odd backslashes)
# r"C:\path\"   <-- SyntaxError! (raw string cannot end with '\')
# Workaround:
path3 = r"C:\path" + "\\"
```

Example 44: f-Strings — Full Feature Tour

```
name  = "Alice"
score = 95.678
items = [10, 20, 30]

# Basic expression
print(f"Name: {name}, Score: {score}")

# Format specification inside {}
print(f"Score: {score:.2f}")          # Score: 95.68
print(f"Score: {score:010.3f}")       # Score: 000095.678  (zero-padded, width 10)
```

```python
print(f"Hex  : {255:#010x}")          # Hex  : 0x000000ff
print(f"Sci  : {0.000123:e}")         # Sci  : 1.230000e-04

# Alignment
print(f"{'left':<10}|{'center':^10}|{'right':>10}")
# left      |  center  |     right

# Debug specifier (Python 3.8+)
x = 42
print(f"{x=}")            # x=42
print(f"{score=:.1f}")    # score=95.7

# Arbitrary expressions
print(f"{len(items)} items, sum={sum(items)}")   # 3 items, sum=60
print(f"{name.upper()!r}")     # 'ALICE'  (!r calls repr())
print(f"{2**10 = }")           # 2**10 = 1024

# Nested braces for dynamic width
width = 15
print(f"{'centered':^{width}}")   # '   centered   '

# Multiline f-string
report = (
    f"Student : {name}\n"
    f"Score   : {score:.1f}\n"
    f"Grade   : {'A' if score >= 90 else 'B'}"
)
print(report)
```

Example 45: Byte Strings and Encoding

```python
# byte string -- type is bytes, not str
b = b"hello"
print(type(b))           # <class 'bytes'>
print(b[0])              # 1box4  (integer, not char)
print(b[0:3])            # b'hel'

% Encoding str -> bytes
s = "Hello, GATE!"
encoded = s.encode("utf-8")          # bytes
print(encoded)                        # b'Hello, GATE!'

% Decoding bytes -> str
decoded = encoded.decode("utf-8")
print(decoded)                        # Hello, GATE!
```

```python
# Non-ASCII characters
greek = "Λλϕα"
b2 = greek.encode("utf-8")
print(b2)               # b'\xce\x91\xce\xbb\xce\xb8\xce\xb1'
print(len(greek))       # 4 characters
print(len(b2))          # 8 bytes (each Greek char = 2 UTF-8 bytes)
```

2.1.3 Slicing

2.1.3.1 The `s[start:stop:step]` Syntax

String Slicing — Formal Definition

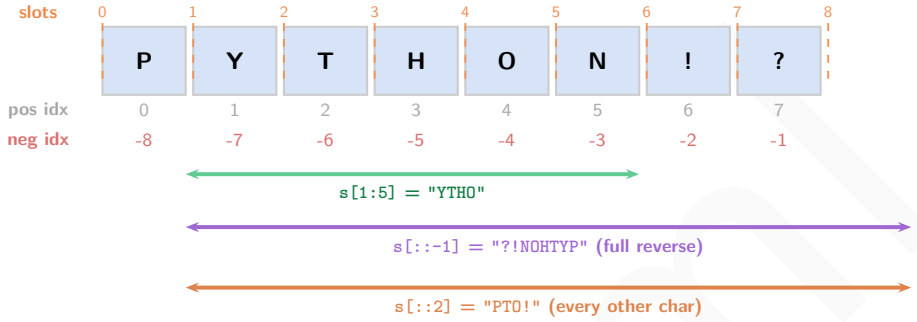
Slicing extracts a substring using the syntax:

`s[start : stop : step]`

- Returns a **new string** (original unchanged)

- `start`: index of first included character (default: `0` for positive step, `len(s)-1` for negative step)
- `stop`: index of first *excluded* character (default: `len(s)` for positive step, `-len(s)-1` for negative step)
- `step`: stride (default: `1`); negative step reverses direction
- Out-of-range indices are *clamped* (no `IndexError` on slices)

Characters included: indices $\{start,\ start + step,\ start + 2 \cdot step,\ \ldots\}$ such that the index is in $[start, stop)$ for positive step, or $(stop, start]$ for negative step.



Slicing Rules and Special Cases

- **Omitted start**: defaults to beginning (`s[:3]` = first 3 chars)
- **Omitted stop**: defaults to end (`s[2:]` = from index 2 onward)
- **Omitted both**: `s[:]` is a *shallow copy* (new object, same content)
- **Negative step**: `s[a:b:-1]` goes from a down to b+1; `s[::-1]` reverses entire string
- **Out-of-bounds silently clamped**: `s[0:1000]` is same as `s[0:len(s)]`; no `IndexError`
- **Empty result**: step-incompatible range returns `""`
- **Step must not be 0**: `s[::0]` raises `ValueError`
- Negative indices in slicing: `s[-3:-1]` is second-last two chars

Example 46: Slicing — All Patterns

```
s = "PYTHON!?"

# Basic slices
print(s[1:5])       # YTHO    (chars at index 1,2,3,4)
print(s[:3])        # PYT     (first 3)
print(s[3:])        # HON!?   (from index 3 to end)
print(s[:])         # PYTHON!?  (full copy)

# Negative indices in slices
print(s[-3:])       # N!?     (last 3 characters)
print(s[:-2])       # PYTHON   (all except last 2)
print(s[-5:-2])     # HON     (slice within negative range)

# Step
print(s[::2])       # PTO!    (every 2nd char: P,T,O,!)
print(s[::3])       # PH?     (every 3rd: P,H,?)
print(s[1::2])      # YHN     (start at 1, every 2nd)

# Reversal
print(s[::-1])      # ?!NOHTYP (completely reversed)
print(s[5:1:-1])    # NOHY    (from 5 down to 2)
```

```python
print(s[::-2])      # ?!TP   (reverse every 2nd)

# Out-of-bounds -- no error
print(s[0:1000])    # PYTHON!?  (clamped to len)
print(s[100:200])   # ""  (empty, no error)

# Common patterns
def is_palindrome(t): return t == t[::-1]
print(is_palindrome("racecar"))  # True
print(is_palindrome("python"))   # False

# Remove prefix/suffix (pre-Python 3.9)
url = "https://example.com"
if url.startswith("https://"):
    url = url[len("https://"):]  # slicing off prefix
print(url)    # example.com
```

2.1.4 Key String Methods

2.1.4.1 Searching & Testing

Searching and Testing Methods

Method	Behaviour
`find(sub[, start, end])`	Returns lowest index of `sub`, or `-1`
`rfind(sub[, start, end])`	Returns highest (rightmost) index, or `-1`
`index(sub)`	Like `find` but raises `ValueError` if not found
`rindex(sub)`	Like `rfind` but raises `ValueError`
`count(sub[, start, end])`	Count non-overlapping occurrences
`startswith(prefix[,s,e])`	`True` if string begins with `prefix`
`endswith(suffix[,s,e])`	`True` if string ends with `suffix`
`in` operator	`"py" in "python"` → `True` (substring search)
Character testing (return `True`/`False`):	
`isalpha()`	All alphabetic
`isdigit()`	All digit characters
`isnumeric()`	All numeric (includes Unicode numerals)
`isalnum()`	All alphanumeric
`isspace()`	All whitespace
`isupper()`	All cased chars are uppercase
`islower()`	All cased chars are lowercase
`istitle()`	Title-case (each word capitalised)

find vs index — Key Differences

- `find()` returns `-1` when not found — safe for conditional checks without try/except
- `index()` raises `ValueError` when not found — use when absence of the substring is a *programming error*
- Both accept optional `start` and `end` arguments to restrict the search range (like a slice, but doesn't create a new string)
- `in` is the idiomatic membership test when you only need existence (not position) — most readable
- `count()` counts *non-overlapping* occurrences: `"aaa".count("aa") == 1` (not 2)

Example 47: Searching Methods — find, count, startswith

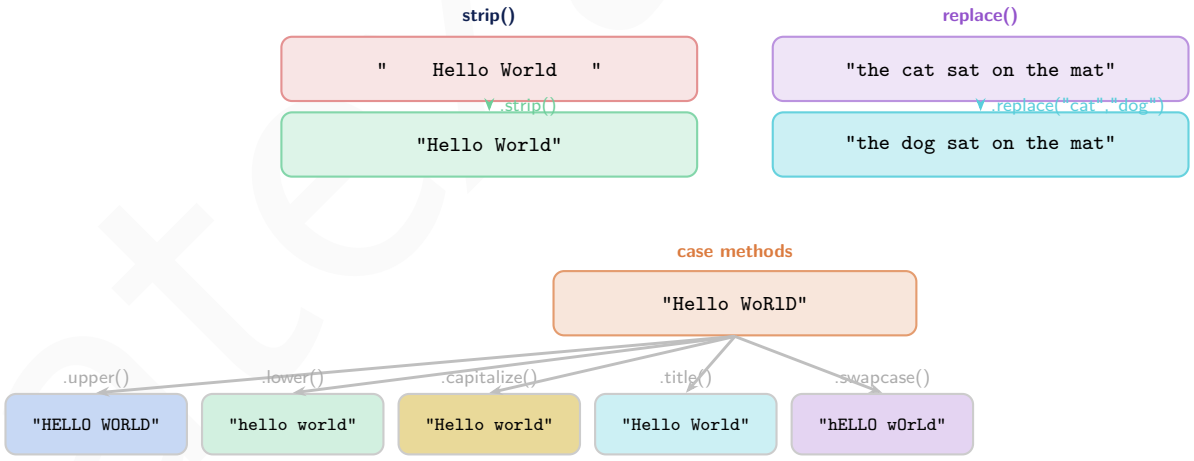
```
s = "the quick brown fox jumps over the lazy dog"

# find / rfind
print(s.find("the"))            # 0  (first occurrence)
print(s.rfind("the"))           # 31 (last occurrence)
print(s.find("cat"))            # -1 (not found)
print(s.find("the", 5))         # 31 (search from index 5)
print(s.find("the", 5, 20))     # -1 (search only s[5:20])

# index raises ValueError
try:
    s.index("cat")
except ValueError as e:
    print(e)   # substring not found

# count (non-overlapping)
print(s.count("the"))           # 2
print("aaaaaa".count("aa"))     # 3  (non-overlapping: aa|aa|aa)
print("aaa".count("aa"))        # 1  (aa|a -- only 1 non-overlapping)

# startswith / endswith with tuple
files = ["report.pdf", "data.csv", "image.png", "notes.txt"]
docs  = [f for f in files if f.endswith((".pdf", ".txt"))]
print(docs)   # ['report.pdf', 'notes.txt']

# in operator
print("fox" in s)       # True
print("FOX" in s)       # False  (case-sensitive)
print("" in s)          # True   (empty string is in any string)

# Character testing
print("Python3".isalpha())   # False (digit present)
print("Python".isalpha())    # True
print("12345".isdigit())     # True
print("abc123".isalnum())    # True
print("  \t\n".isspace())    # True
print("GATE".isupper())      # True
print("Gate Exam".istitle()) # True
```

2.1.4.2 Modification Methods (Return New String)

String Modification Methods

All modification methods return **new string objects**; the original is unchanged.

Method	Behaviour
`replace(old, new[, count])`	Replace all (or `count`) occurrences
`strip([chars])`	Remove leading+trailing whitespace (or given chars)
`lstrip([chars])`	Remove leading only
`rstrip([chars])`	Remove trailing only
`upper()`	All uppercase
`lower()`	All lowercase
`capitalize()`	First char upper, rest lower
`title()`	First char of each word upper
`swapcase()`	Swap upper ↔ lower
`casefold()`	Aggressive lowercase (Unicode-aware; for comparisons)
`zfill(width)`	Pad with zeros on the left
`center(w[, fill])`	Centre in field of width `w`
`ljust(w[, fill])`	Left-justify in field of width `w`
`rjust(w[, fill])`	Right-justify
`expandtabs(ts)`	Replace tabs with spaces (`ts`=tabsize, default 8)



Example 48: strip, replace, case, justify — All Forms

```python
# strip variants
s = "   hello world   "
print(repr(s.strip()))    # 'hello world'
print(repr(s.lstrip()))   # 'hello world   '
print(repr(s.rstrip()))   # '   hello world'

# strip with character set
t = "***ERROR: file not found***"
print(t.strip("*"))       # 'ERROR: file not found'
print(t.strip("*E"))      # 'RROR: file not found'   (strips any of *, E)
```

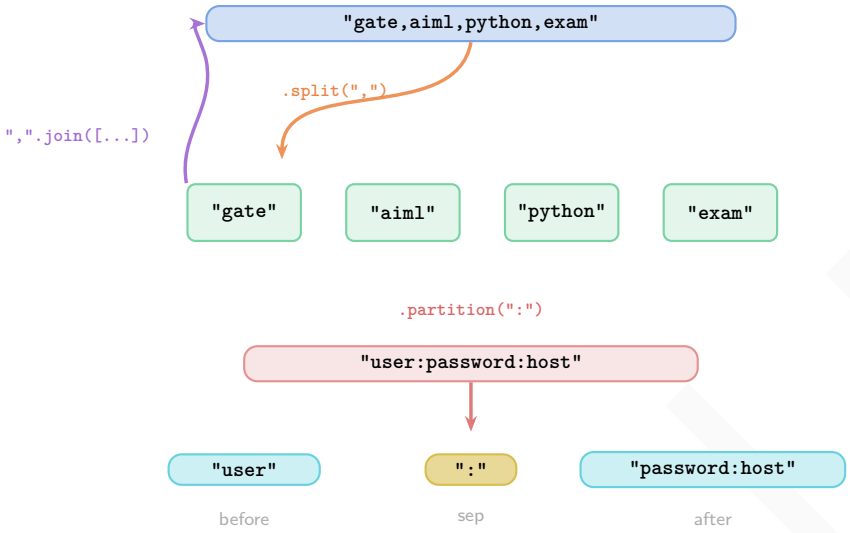
```python

# replace
sentence = "the cat sat on the mat"
print(sentence.replace("at", "og"))          # the cog sog on the mog
print(sentence.replace("the", "a", 1))       # a cat sat on the mat (count=1)

# Case methods
s = "hello WoRlD"
print(s.upper())         # HELLO WORLD
print(s.lower())         # hello world
print(s.capitalize())    # Hello world  (only first char up)
print(s.title())         # Hello World  (each word)
print(s.swapcase())      # HELLO wOrLd

# casefold for international comparison
print("stra e".casefold())      # strasse  (German    -> ss)
print("stra e".lower())         # stra e   (lower doesn't expand   )
print("stra e".casefold() == "strasse")  # True (correct comparison)

# Justification and fill
print("42".zfill(6))            # 000042
print("-42".zfill(6))           # -00042  (sign preserved)
print("GATE".center(12, '-'))   # ----GATE----
print("left".ljust(10, '.'))    # left......
print("right".rjust(10, '.'))   # .....right

# Method chaining (each returns new string)
result = "  GATE Exam 2025  ".strip().lower().replace(" ", "_")
print(result)    # gate_exam_2025
```

2.1.4.3 Splitting & Joining

split, join, partition — Definitions

Method	Behaviour
`split(sep=None, maxsplit=-1)`	Split on sep; `None` splits on *any* whitespace and strips empties
`rsplit(sep, maxsplit)`	Like `split` but splits from the right
`splitlines([keepends])`	Split on line boundaries (`\n`, `\r\n`, `\r`, ...)
`"sep".join(iterable)`	Join elements of iterable with separator (all elements must be `str`)
`partition(sep)`	Returns `(before, sep, after)`; if not found: `(whole, "", "")`
`rpartition(sep)`	Like `partition` but finds *last* occurrence

Golden rule for building strings from many pieces:
Never use += in a loop (each iteration creates a new object, $O(n^2)$).
Always use `"".join(list)` ($O(n)$).



Example 49: split and join — All Variants

```python
# Default split: any whitespace, strips leading/trailing
s = "  one  two   three  "
print(s.split())              # ['one', 'two', 'three']

# Split on specific separator
csv = "gate,aiml,python,exam"
print(csv.split(","))         # ['gate', 'aiml', 'python', 'exam']

# maxsplit: limit number of splits
print(csv.split(",", 2))      # ['gate', 'aiml', 'python,exam']

# rsplit: split from right
print(csv.rsplit(",", 1))     # ['gate,aiml,python', 'exam']

# splitlines
text = "Line1\nLine2\r\nLine3\rLine4"
print(text.splitlines())      # ['Line1', 'Line2', 'Line3', 'Line4']
print(text.splitlines(True)) # keeps line endings: ['Line1\n', ...]

# join -- the correct way to build strings
words = ["GATE", "2025", "AI", "ML"]
print(" ".join(words))     # GATE 2025 AI ML
print("-".join(words))     # GATE-2025-AI-ML
print("".join(words))      # GATE2025AIML

# Efficient string building (join vs +=)
parts = []
for i in range(5):
    parts.append(str(i * i))   # collect first
result = ", ".join(parts)      # join once at the end
print(result)   # 0, 1, 4, 9, 16

# partition
conn = "user:password:hostname"
before, sep, after = conn.partition(":")
print(before, sep, after)   # user : password:hostname

# rpartition (last occurrence)
before2, sep2, after2 = conn.rpartition(":")
print(before2, sep2, after2) # user:password : hostname

# partition when sep not found
a, b, c = "nocohere".partition(":")
print(repr(a), repr(b), repr(c))  # 'nocohere' '' ''
```

2.1.4.4 String Formatting

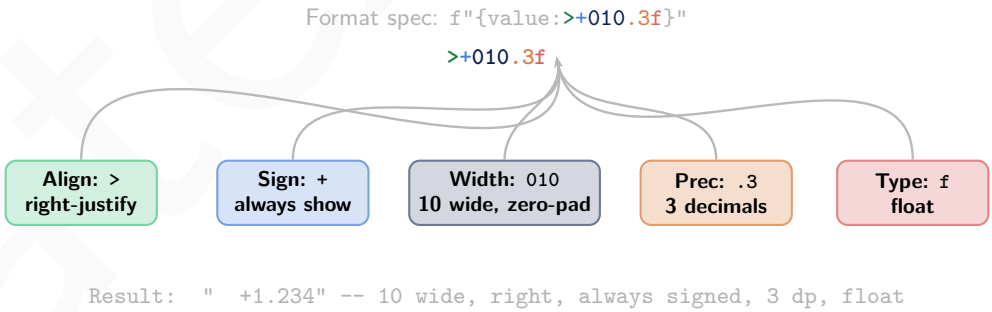
Three Generations of Python String Formatting

Style	Syntax	Python Version
%-formatting (old)	`"%d items at %.2f" % (n, p)`	All
`str.format()` (new)	`"{0} items at {1:.2f}".format(n,p)`	2.6+
f-string (modern)	`f"{n} items at {p:.2f}"`	3.6+

Format specification mini-language (same for `format()` and f-strings):

`[[fill]align][sign][z][#][0][width][grouping_option][.precision][type]`

Component	Values		
align	< left, > right, ^ centre, = sign-aware		
sign	+ always, − negatives only, space		
#	Alternate form (0b, 0o, 0x prefix)		
0	Zero-pad within width		
width	Minimum field width		
.prec	Float decimal places; str max chars		
type	d int	f float	e scientific
	s str	x/X hex	o octal
	b binary	g general	% percentage



Format Type Codes Reference

- `d` — Decimal integer: `f"{255:d}"` → 255
- `b` — Binary: `f"{10:#b}"` → 0b1010
- `o` — Octal: `f"{8:#o}"` → 0o10
- `x/X` — Hex lower/upper: `f"{255:#x}"` → 0xff
- `f` — Fixed float: `f"{3.14159:.2f}"` → 3.14

- **e/E** — Scientific: `f"{0.00123:e}"` → `1.230000e-03`
- **g** — General (shorter of f/e): `f"{0.00001:g}"` → `1e-05`
- **%** — Percentage: `f"{0.856:.1%}"` → `85.6%`
- **s** — String (default): `f"{"gate":>10s}"` → `"      gate"`
- **,** — Thousands separator: `f"{1234567:,}"` → `1,234,567`
- **_** — Underscore separator (3.6+): `f"{1234567:_}"` → `1_234_567`

Example 50: % Formatting (Old Style)

```
name  = "Alice"
score = 95.678
rank  = 3

# Basic substitution
print("Name: %s, Score: %.2f" % (name, score))
# Name: Alice, Score: 95.68

# Width and alignment
print("%-10s | %5d | %8.3f" % (name, rank, score))
# Alice      |     3 |   95.678

# Multiple types
for item, qty, price in [("Pen",5,1.5),("Book",2,299.0)]:
    print("%-8s %3d  Rs %7.2f" % (item, qty, qty*price))
# Pen          5  Rs    7.50
# Book         2  Rs  598.00

# Named substitution using dict
print("%(name)s scored %(score).1f" % {"name":name, "score":score})
# Alice scored 95.7
```

Example 51: `str.format()` (New Style)

```
# Positional
print("{} items at Rs {:.2f} each".format(5, 49.99))
# 5 items at Rs 49.99 each

# Explicit positional
print("{0} and {1}, then {0} again".format("A","B"))
# A and B, then A again

# Named
print("{name} scored {score:.1f}%".format(name="Alice", score=95.7))
# Alice scored 95.7%

# Alignment and fill
print("{:*^20}".format("GATE"))    # ********GATE********
print("{:>10}".format("right"))    # "     right"
print("{:<10}".format("left"))     # "left      "

# Number formatting
print("{:010.3f}".format(3.14))    # 000003.140
print("{:+.2f}".format(-42.5))     # -42.50
print("{:+.2f}".format(42.5))      # +42.50
print("{:,}".format(1_000_000))    # 1,000,000
print("{:#010x}".format(255))      # 0x000000ff
print("{:#010b}".format(10))       # 0b00001010

# Dynamic width/precision
width, prec = 12, 4
```

```python
print("{:{w}.{p}f}".format(3.14159, w=width, p=prec))
# "        3.1416"
```

Example 52: f-Strings — Modern Formatting (Full Spec)

```python
import math

pi    = math.pi
n     = 1_000_000
name  = "GATE DA"
score = 0.8567

# Basic
print(f"{name} pass rate: {score:.1%}")   # GATE DA pass rate: 85.7%

# Alignment
for lang, rank in [("Python",1),("Java",2),("C",3)]:
    print(f"{lang:<10} rank: {rank:>3}")
# Python     rank:   1
# Java       rank:   2
# C          rank:   3

# Number bases
n2 = 255
print(f"Dec:{n2:d}  Bin:{n2:#010b}  Oct:{n2:#o}  Hex:{n2:#x}")
# Dec:255  Bin:0b11111111  Oct:0o377  Hex:0xff

# Scientific and general
print(f"pi = {pi:.5f}")          # pi = 3.14159
print(f"pi = {pi:.3e}")          # pi = 3.142e+00
print(f"pi = {pi:.4g}")          # pi = 3.142
print(f"big = {n:_}")            # big = 1_000_000
print(f"big = {n:,}")            # big = 1,000,000

# Debug with =
x, y = 10, 20
print(f"{x=}, {y=}, {x+y=}")     # x=10, y=20, x+y=30

# Expression inside format spec
col_width = 8
headers = ["Name", "Score", "Rank"]
print("  ".join(f"{h:^{col_width}}" for h in headers))
#    Name      Score    Rank

# Nested f-string
fmt = ".3f"
print(f"{pi:{fmt}}")             # 3.142

# Conversion flags: !r (repr), !s (str), !a (ascii)
s = "caf "
print(f"{s!r}")      # 'caf '
print(f"{s!a}")      # 'caf\xe9'  (ascii-safe)
```

Example 53: Comparison of All Three Formatting Styles

```python
name, score, rank = "Alice", 95.678, 1

# Old style
print("%-10s %5.2f #%d"    % (name, score, rank))

# .format()
print("{:<10} {:5.2f} #{:d}".format(name, score, rank))

# f-string (recommended)
print(f"{name:<10} {score:5.2f} #{rank:d}")
```

```
11
12 # All produce:   Alice      95.68 #1
13
14 # template strings (safe for user input, no arbitrary expressions)
15 from string import Template
16 t = Template("Hello $name, your score is $score")
17 print(t.substitute(name=name, score=round(score,1)))
18 # Hello Alice, your score is 95.7
```

2.1.4.5 Performance and Common Patterns

String Performance Tips and Idiomatic Patterns

- **Concatenation in loop**: use `list.append()` + `"".join()` instead of `s += part` (avoids $O(n^2)$ copying)
- **Check prefix/suffix**: Python 3.9+ has `str.removeprefix()` and `str.removesuffix()` (cleaner than slicing)
- **Palindrome check**: `s == s[::-1]` (simple); for case-insensitive: `s.lower() == s[::-1].lower()`
- **Count words**: `len(s.split())` (handles multiple spaces)
- **Reverse words**: `" ".join(s.split()[::-1])`
- **String to list of chars**: `list(s)` or `[c for c in s]`
- **Check all chars**: `all(c.isdigit() for c in s)`
- **Caesar cipher**: `chr((ord(c) - ord('a') + n) % 26 + ord('a'))`
- **String interning**: `sys.intern(s)` forces a single shared copy of `s` in memory (speeds up repeated equality tests)

Example 54: Useful String Patterns for GATE

```
1  # 1. Count vowels
2  s = "Python Programming"
3  vowels = sum(1 for c in s.lower() if c in "aeiou")
4  print(vowels)    # 5
5
6  # 2. Reverse words in a sentence
7  sentence = "  Hello World  Python  "
8  reversed_words = " ".join(sentence.split()[::-1])
9  print(reversed_words)   # Python World Hello
10
11 # 3. Check pangram (contains all 26 letters)
12 def is_pangram(text):
13     return set('abcdefghijklmnopqrstuvwxyz').issubset(set(text.lower()))
14 print(is_pangram("The quick brown fox jumps over the lazy dog"))  # True
15
16 # 4. Remove duplicates preserving order
17 s = "programming"
18 unique = "".join(dict.fromkeys(s))
19 print(unique)    # progamin
```

2.2 Lists

2.2.1 List Properties

2.2.1.1 Core Characteristics

List — Definition and Core Properties

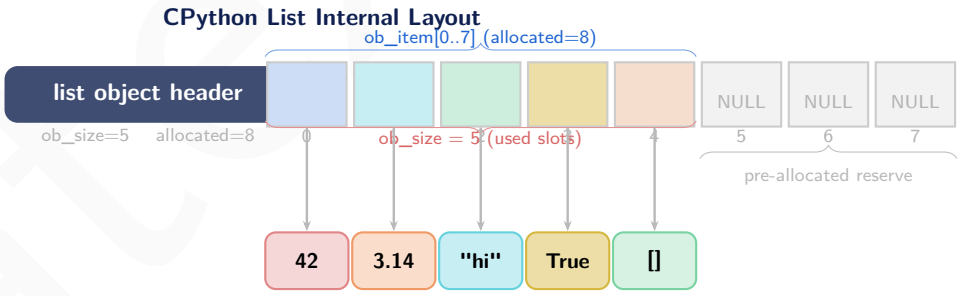
A Python **list** is an **ordered, mutable, heterogeneous sequence** that allows duplicate elements. It is Python's workhorse general-purpose container.

Property	Meaning
Ordered	Elements maintain insertion order; index-accessible
Mutable	Elements can be added, removed, or replaced in place
Heterogeneous	Can hold objects of *different* types simultaneously
Allows duplicates	`[1, 1, 2, 2, 3]` is a valid list
Dynamic size	Grows and shrinks automatically; no fixed capacity

Dynamic Array Implementation

Internally CPython implements `list` as a **dynamic array** (array of pointers to PyObjects):

- A contiguous block of memory stores *pointers* (references) to objects, not the objects themselves
- When the array is full, Python **over-allocates** a larger block (roughly $\times 1.125$ growth factor) and copies pointers over — this gives **amortised** $O(1)$ for `append`
- Random access `L[i]` is $O(1)$ (pointer arithmetic)
- Insertion / deletion at the middle is $O(n)$ (shifting pointers)



List Properties Summary

- **Heterogeneous**: `[1, "two", 3.0, True, None, [5,6]]` is valid
- **Nestable**: lists can contain other lists (any depth)
- **Mutable**: supports item assignment `L[i] = x`, slice assignment `L[a:b] = [...]`, `del L[i]`
- **Ordered**: `[1,2,3] != [3,2,1]` even with same elements
- **Iterable**: works with `for`, `map`, `filter`, `zip`, comprehensions
- **Hashable?** No — lists are mutable so cannot be dict keys or set members
- **Truthiness**: empty list `[]` is falsy; any non-empty list is truthy

2.2.2 List Creation

2.2.2.1 All Creation Methods

Ways to Create a List

Method	Syntax / Example
Literal	`[1, 2, 3]`, `[]` (empty)
Constructor	`list(iterable)`
Repetition	`[0] * 5` → `[0,0,0,0,0]`
List comprehension	`[x**2 for x in range(5)]`
`range`	`list(range(1, 11))`
`split`	`"a b c".split()` → `['a','b','c']`

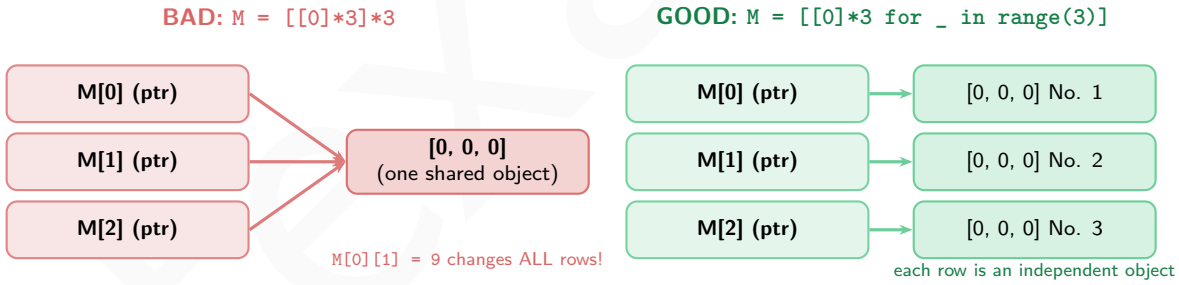
Repetition Pitfall — Shared Inner Lists

The expression `[[]] * n` creates a list of `n` references to the **same single inner list object**. Modifying one element modifies all of them.

`M = [[0]*3]*3` ⇒ all three rows point to the *same* list

Correct pattern for independent rows:

```
M = [[0]*3 for _ in range(3)]
```



Example 55: List Creation — All Methods

```
# Literal
a = [1, 2, 3]
b = []                          # empty list
c = [1, "hello", 3.14, True, None]  # heterogeneous

# Constructor from iterable
d = list("GATE")                # ['G', 'A', 'T', 'E']
e = list(range(1, 6))           # [1, 2, 3, 4, 5]
f = list((10, 20, 30))          # from tuple
g = list({1, 2, 3})             # from set (order not guaranteed)

# Repetition
zeros  = [0] * 5                # [0, 0, 0, 0, 0]
filler = [None] * 4             # [None, None, None, None]

# PITFALL: nested mutable
bad = [[]] * 3
bad[0].append(1)
```

```python
print(bad)   # [[1], [1], [1]]  -- ALL rows affected!

# CORRECT: comprehension
good = [[] for _ in range(3)]
good[0].append(1)
print(good)  # [[1], [], []]   -- only row 0 affected

# 2D matrix (3x4) of zeros
matrix = [[0] * 4 for _ in range(3)]
matrix[1][2] = 99
print(matrix)
# [[0, 0, 0, 0],
#  [0, 0, 99, 0],
#  [0, 0, 0, 0]]
```

2.2.3 Indexing & Slicing

2.2.3.1 List Indexing, Slice Assignment and `del`

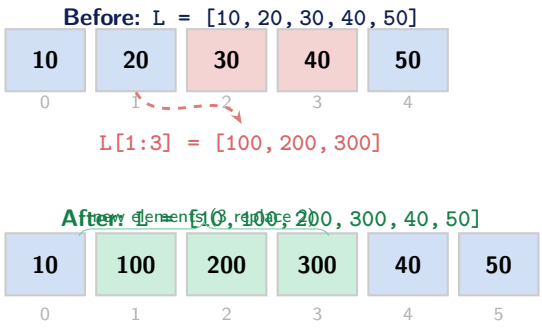
List Slicing and Slice Assignment

List slicing uses the same `L[start:stop:step]` syntax as strings, but because lists are **mutable**, they additionally support:

- **Slice assignment**: `L[a:b] = [x, y, z]` — replaces the slice with new elements; can *change the length* of the list
- **`del` statement**: `del L[i]` removes one element; `del L[a:b]` removes a range; `del L[::2]` removes every other
- Slicing returns a **new list** (shallow copy of that slice)

Key differences from string slicing:

String	List
`s[a:b] = ...` → TypeError	`L[a:b] = [...]` → in-place replace
Immutable: no `del s[i]`	`del L[i]` works
Slice is a new `str`	Slice is a new `list` (shallow copy)



Example 56: Slice Assignment and `del` — Changing List Length

```
L = [10, 20, 30, 40, 50]

% Regular slicing: returns NEW list (no modification)
```

```
sub = L[1:4]
print(sub)        # [20, 30, 40]
print(L)          # [10, 20, 30, 40, 50]  unchanged

% Slice assignment: modifies IN PLACE, can change length
L[1:3] = [100, 200, 300]   # replace 2 elements with 3
print(L)          # [10, 100, 200, 300, 40, 50]  (length 6)

L[1:4] = []       # delete elements at index 1,2,3
print(L)          # [10, 40, 50]  (length 3)

L[1:1] = [99, 88]  # insert without replacing (stop==start)
print(L)          # [10, 99, 88, 40, 50]

# del statement
L = [10, 20, 30, 40, 50]
del L[2]          # delete element at index 2
print(L)          # [10, 20, 40, 50]

del L[1:3]        # delete slice
print(L)          # [10, 50]

L = list(range(10))
del L[::2]        # delete every other element (indices 0,2,4,6,8)
print(L)          # [1, 3, 5, 7, 9]

# Negative index assignment
L = [1, 2, 3, 4, 5]
L[-1] = 99
print(L)          # [1, 2, 3, 4, 99]
```

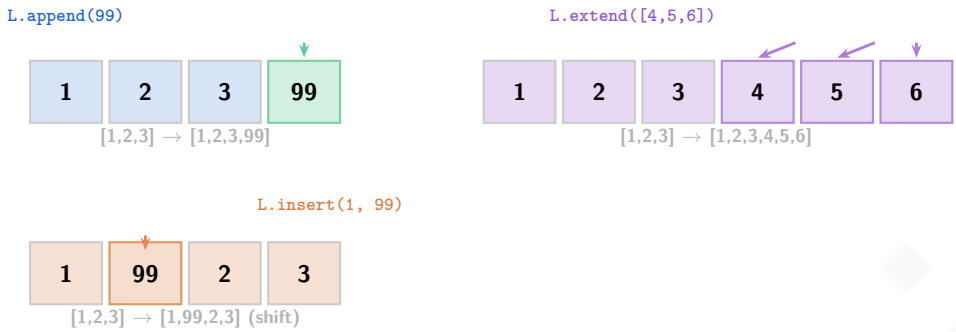
2.2.4 List Methods

2.2.4.1 Mutation Methods (In-Place)

Mutating Methods — All Return `None`

All in-place mutation methods **return** `None` (not the list). This is a common bug: `L = L.sort()` sets `L` to `None`!

Method	Complexity	Behaviour
`append(x)`	$O(1)$ amortised	Add `x` as single element at end
`extend(iterable)`	$O(k)$	Add all elements of iterable at end
`insert(i, x)`	$O(n)$	Insert `x` before index `i`
`remove(x)`	$O(n)$	Remove *first* occurrence of `x`; `ValueError` if absent
`pop()`	$O(1)$	Remove and return last element
`pop(i)`	$O(n)$	Remove and return element at index `i`
`clear()`	$O(n)$	Remove all elements
`sort(key, reverse)`	$O(n \log n)$	Timsort; stable; in-place
`reverse()`	$O(n)$	Reverse in place



sort() Properties — Timsort

- **Timsort**: hybrid merge sort + insertion sort; Python's native sort since Python 2.3
- **Stable**: equal elements keep their relative order
- **In-place**: modifies the list; returns `None`
- **`key` parameter**: a function applied to each element before comparison; the original values are stored unmodified
- **`reverse=True`**: sorts in descending order
- **`sorted(L)` vs `L.sort()`**: `sorted()` returns a *new* sorted list; `L.sort()` is in-place
- Time: $O(n \log n)$ worst/average; $O(n)$ best case (already sorted runs)

Example 57: append, extend, insert, remove, pop

```
L = [1, 2, 3]

# append: adds one object (even if it's a list)
L.append(4)
print(L)                # [1, 2, 3, 4]
L.append([5, 6])        # adds [5,6] as a SINGLE element (nested list)
print(L)                # [1, 2, 3, 4, [5, 6]]

# extend: adds each element of the iterable
L = [1, 2, 3]
L.extend([4, 5, 6])     # unpacks and appends individually
print(L)                # [1, 2, 3, 4, 5, 6]
L.extend("AB")          # strings are iterable
print(L)                # [1, 2, 3, 4, 5, 6, 'A', 'B']

# insert: O(n)
L = [10, 20, 30, 40]
L.insert(0, 99)         # insert at beginning
print(L)                # [99, 10, 20, 30, 40]
L.insert(len(L), 0)     # same as append
print(L)                # [99, 10, 20, 30, 40, 0]
L.insert(-1, 77)        # before last element
print(L)                # [99, 10, 20, 30, 40, 77, 0]

# remove: removes FIRST occurrence
L = [3, 1, 4, 1, 5, 9, 2, 6]
L.remove(1)             # removes first 1
print(L)                # [3, 4, 1, 5, 9, 2, 6]
try:
    L.remove(99)        # raises ValueError if not found
except ValueError as e:
    print(e)            # list.remove(x): x not in list

# pop: removes and returns
L = [10, 20, 30, 40, 50]
```

```
print(L.pop())          # 50  (from end, O(1))
print(L.pop(1))         # 20  (from index 1, O(n))
print(L)                # [10, 30, 40]
```

Example 58: `sort()` vs `sorted()` — key and reverse

```
# sort() -- in-place, returns None
nums = [3, 1, 4, 1, 5, 9, 2, 6]
nums.sort()
print(nums)             # [1, 1, 2, 3, 4, 5, 6, 9]

# sorted() -- returns new list, original unchanged
nums2 = [3, 1, 4, 1, 5, 9, 2, 6]
result = sorted(nums2)
print(result)           # [1, 1, 2, 3, 4, 5, 6, 9]
print(nums2)            # [3, 1, 4, 1, 5, 9, 2, 6]  unchanged!

# reverse=True
nums.sort(reverse=True)
print(nums)             # [9, 6, 5, 4, 3, 2, 1, 1]

# key function -- sort by absolute value
data = [-5, 2, -1, 8, -3]
data.sort(key=abs)
print(data)             # [-1, 2, -3, -5, 8]

# sort strings by length
words = ["banana", "kiwi", "apple", "fig", "pear"]
words.sort(key=len)
print(words)            # ['fig', 'kiwi', 'pear', 'apple', 'banana']

# sort complex objects
students = [("Alice",85), ("Bob",92), ("Charlie",78)]
students.sort(key=lambda s: s[1], reverse=True)
print(students)   # [('Bob',92), ('Alice',85), ('Charlie',78)]

# stable sort: equal keys preserve original order
data = [("a",2),("b",1),("c",2),("d",1)]
data.sort(key=lambda x: x[1])
print(data)   # [('b',1), ('d',1), ('a',2), ('c',2)]

# Common bug
L = [3, 1, 2]
L = L.sort()    # BUG: sort() returns None!
print(L)        # None
```

2.2.4.2 Non-Mutating Methods

Non-Mutating List Methods

Method	Returns	Behaviour
`index(x[, start, end])`	`int`	Index of first `x`; `ValueError` if absent
`count(x)`	`int`	Number of occurrences of `x`
`copy()`	`list`	Shallow copy (same as `L[:]`)
Built-in functions (work on any iterable):		
`len(L)`	`int`	Number of elements
`min(L)`	element	Smallest element
`max(L)`	element	Largest element
`sum(L)`	number	Sum (default start=0)
`sorted(L)`	new list	Sorted copy
`reversed(L)`	iterator	Reverse iterator
`enumerate(L)`	iterator	`(index, value)` pairs
`zip(L1, L2)`	iterator	Paired tuples

Example 59: index, count, enumerate, zip

```
L = [10, 20, 30, 20, 40, 20]

# index: first occurrence
print(L.index(20))         # 1
print(L.index(20, 2))      # 3  (search from index 2)
print(L.index(20, 4))      # 5  (search from index 4)
try:
    L.index(99)
except ValueError as e:
    print(e)               # 99 is not in list

# count
print(L.count(20))         # 3
print(L.count(99))         # 0  (no error for missing)

# enumerate: index + value pairs
fruits = ["apple", "banana", "cherry"]
for i, fruit in enumerate(fruits):
    print(f"{i}: {fruit}")

for i, fruit in enumerate(fruits, start=1):  # start index at 1
    print(f"{i}. {fruit}")

# zip: pair corresponding elements
names  = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]
for name, score in zip(names, scores):
    print(f"{name}: {score}")

# zip stops at shortest
a = [1, 2, 3, 4]
b = [10, 20]
print(list(zip(a, b)))     # [(1,10), (2,20)]

```

```python
# zip(*list) to unzip (transpose)
pairs = [(1,'a'), (2,'b'), (3,'c')]
nums, chars = zip(*pairs)
print(nums)    # (1, 2, 3)
print(chars)   # ('a', 'b', 'c')
```

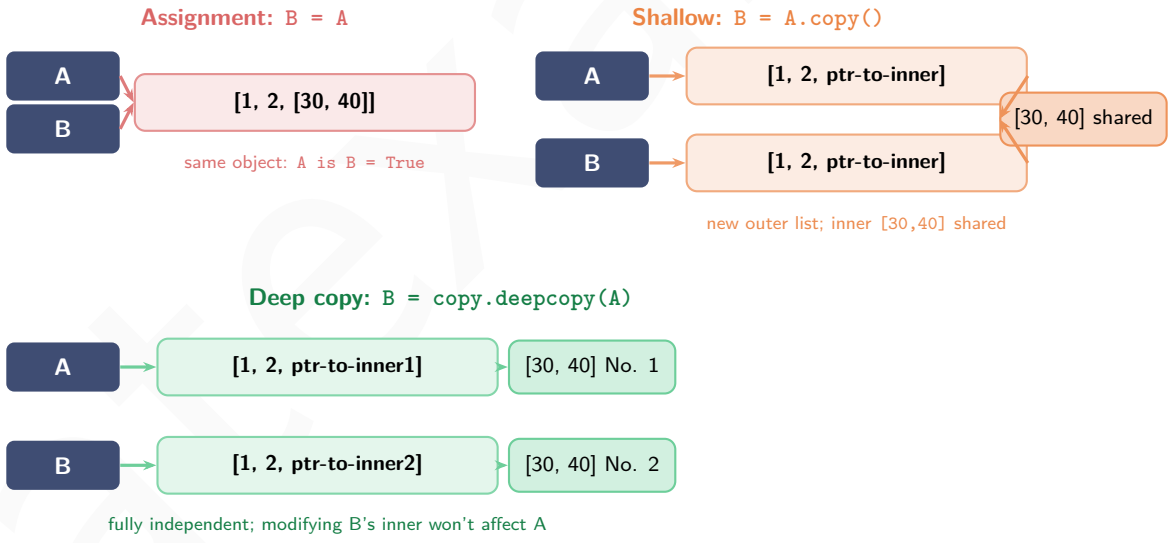
2.2.5 Copying Semantics

2.2.5.1 Assignment, Shallow Copy, Deep Copy

Three Levels of List Copying

Operation	Method	New List?	Inner Objects?
Assignment	`B = A`	No	Same (alias)
Shallow	`B = A.copy()`	Yes	Shared (same objects)
	`B = A[:]`	Yes	Shared
	`B = list(A)`	Yes	Shared
Deep	`copy.deepcopy(A)`	Yes	Fully independent

Shallow copy is sufficient when the list contains only *immutable* objects (ints, strings, tuples). A **deep copy** is needed when the list contains *mutable* inner objects (lists, dicts, custom objects).



Example 60: Assignment vs Shallow vs Deep Copy

```python
import copy

original = [1, 2, [30, 40]]

#         1. Assignment: alias, NOT a copy

alias = original
alias.append(99)
print(original)   # [1, 2, [30, 40], 99]
alias[2].append(50)
print(original)   # [1, 2, [30, 40, 50], 99]
```

```
11
12 #         2. Shallow copy


13 original = [1, 2, [30, 40]]
14 shallow = original.copy()
15
16 shallow.append(99)           # does NOT affect original
17 print(original)   # [1, 2, [30, 40]]
18
19 shallow[2].append(50)        # DOES affect original!
20 print(original)   # [1, 2, [30, 40, 50]]
21 print(shallow)    # [1, 2, [30, 40, 50], 99]
22
23 #         3. Deep copy: fully independent

24 original = [1, 2, [30, 40]]
25 deep = copy.deepcopy(original)
26
27 deep.append(99)
28 deep[2].append(50)           # does NOT affect original
29 print(original)   # [1, 2, [30, 40]]
30 print(deep)       # [1, 2, [30, 40, 50], 99]
31
32 #         Confirming identity


33 a = [1, [2, 3]]
34 s = a.copy()
35 d = copy.deepcopy(a)
36
37 print(a is s)            # False (different outer list)
38 print(a[1] is s[1])      # True  (same inner list -- shallow!)
39 print(a[1] is d[1])      # False (different inner list -- deep)
```

Example 61: Nested List Pitfall — `[[0]*3]*3`

```
1  # PITFALL: repetition creates shared references to inner list
2  bad_matrix = [[0] * 3] * 3
3  print(bad_matrix)          # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
4  bad_matrix[0][1] = 99
5  print(bad_matrix)
6
7  # Proof: all rows are the same object
8  print(bad_matrix[0] is bad_matrix[1])  # True
9  print(bad_matrix[1] is bad_matrix[2])  # True
10
11 # CORRECT: list comprehension creates independent rows
12 good_matrix = [[0] * 3 for _ in range(3)]
13 good_matrix[0][1] = 99
14 print(good_matrix)
15
16 print(good_matrix[0] is good_matrix[1])  # False
```

2.2.6 append vs extend vs + vs +=

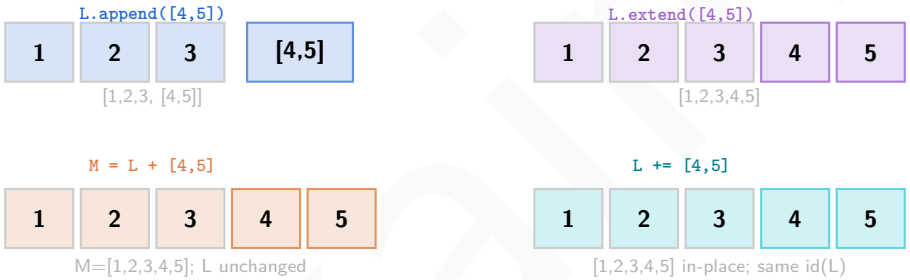
2.2.6.1 Adding Elements and Combining Lists

Four Ways to Add to a List — Differences

Op	In-place?	Adds	Result type	Complexity
`append(x)`	Yes	`x` as one element	`None`	$O(1)$ amort.
`extend(it)`	Yes	Each item of `it`	`None`	$O(k)$
`L + M`	No	Each item of `M`	new list	$O(n+m)$
`L += M`	Yes	Each item of `M`	`None`	$O(k)$

Key distinction:

- `append([4,5])` adds the list `[4,5]` as a *single nested element*
- `extend([4,5])` adds 4 and 5 as *two separate elements*
- `L + [4,5]` creates a *new list*; `L` is unchanged
- `L += [4,5]` calls `__iadd__` (same as `extend`); same id



Example 62: append vs extend vs + vs += — Side by Side

```python
#         append: adds ONE item (even if it's a list)
L = [1, 2, 3]
L.append([4, 5])
print(L)                # [1, 2, 3, [4, 5]]
print(len(L))           # 4

#         extend: unpacks iterable, adds each item

L = [1, 2, 3]
L.extend([4, 5])
print(L)                # [1, 2, 3, 4, 5]
print(len(L))           # 5

#         + creates NEW list


L = [1, 2, 3]
M = L + [4, 5]
print(M)                # [1, 2, 3, 4, 5]
print(L)                # [1, 2, 3]
print(id(L) == id(M))   # False

#         += modifies IN PLACE (same id)

L = [1, 2, 3]
old_id = id(L)
L += [4, 5]
print(L)                # [1, 2, 3, 4, 5]
print(id(L) == old_id) # True

```

```
#         += vs + with immutables


s = "hello"
old_id = id(s)
s += " world"
print(id(s) == old_id) # False

#         Practical: build list efficiently

L = []
L.extend(range(5))      # L = [0,1,2,3,4]

L2 = []
for x in range(5):
    L2.append(x)

matrix_rows = [[1,2],[3,4],[5,6]]
flat1 = []
for row in matrix_rows:
    flat1.extend(row)
flat2 = []
for row in matrix_rows:
    flat2.append(row)
print(flat1)    # [1, 2, 3, 4, 5, 6]
print(flat2)    # [[1, 2], [3, 4], [5, 6]]
```

Example 63: Important List Patterns for GATE

```
# 1. Flatten a 2D list
matrix = [[1,2,3],[4,5,6],[7,8,9]]
flat = [x for row in matrix for x in row]
print(flat)     # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 2. Remove duplicates preserving order
def unique(lst):
    seen = set()
    return [x for x in lst if not (x in seen or seen.add(x))]
print(unique([3,1,4,1,5,9,2,6,5]))    # [3, 1, 4, 5, 9, 2, 6]

# 3. Rotate list by k positions
def rotate(lst, k):
    k %= len(lst)
    return lst[-k:] + lst[:-k]
print(rotate([1,2,3,4,5], 2))    # [4, 5, 1, 2, 3]

# 4. Stack and Queue using list
stack = []
stack.append(1); stack.append(2); stack.append(3)
print(stack.pop())        # 3

from collections import deque
queue = deque()
queue.append(1); queue.append(2); queue.append(3)
print(queue.popleft())    # 1

# 5. Two-pointer technique
def two_sum(lst, target):
    lst_sorted = sorted(lst)
    lo, hi = 0, len(lst_sorted) - 1
    while lo < hi:
        s = lst_sorted[lo] + lst_sorted[hi]
        if s == target:   return (lst_sorted[lo], lst_sorted[hi])
        elif s < target:  lo += 1
        else:             hi -= 1
    return None
print(two_sum([2,7,11,15], 9))    # (2, 7)

```

```python
# 6. List as default argument pitfall
def bad_append(x, L=[]):
    L.append(x)
    return L

print(bad_append(1))    # [1]
print(bad_append(2))    # [1, 2]

def good_append(x, L=None):
    if L is None:
        L = []
    L.append(x)
    return L
```

2.3 Tuples

2.3.1 Tuple Properties

2.3.1.1 Core Characteristics

Tuple — Definition and Core Properties

A Python **tuple** is an **ordered, immutable, heterogeneous sequence** that allows duplicate elements. It is Python's built-in fixed-size container.

Property	Tuple	List (comparison)
Ordered	✓	✓
Mutable	**No**	Yes
Allows duplicates	✓	✓
Heterogeneous	✓	✓
Hashable	✓ (if elements hashable)	**No**
Dict key / set member	✓	**No**
Memory	**Smaller**	Larger (over-allocated)

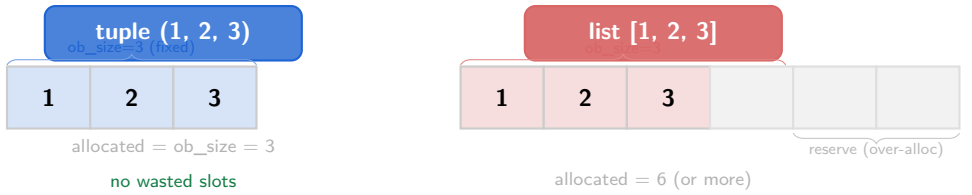
Immutability and Hashability

Immutable means the tuple's structure (which objects it refers to) cannot change after creation. However:

- If a tuple *contains a mutable object* (e.g. a list), that object's *contents* can still change. The tuple's reference to it does not.
- A tuple is **hashable only if all its elements are hashable**. `(1, 2, 3)` is hashable; `(1, [2, 3])` is **not** (list is unhashable).

Formal: `hash(t)` succeeds ⇔ every element `e` in `t` satisfies `hash(e)` succeeds (recursively).
Why tuples are faster / smaller:

- CPython allocates *exactly* the right number of slots (no over-allocation)
- Tuples of literals are stored as **compile-time constants** in bytecode
- CPython maintains a **free-list** of small empty/1-element tuples for reuse
- `sys.getsizeof((1,2,3))` < `sys.getsizeof([1,2,3])`



`sys.getsizeof((1,2,3))` = 72 B vs `sys.getsizeof([1,2,3])` = 88 B (CPython 3.11)

Tuple Use Cases

- **Heterogeneous records**: `person = ("Alice", 30, "Delhi")` — fields have fixed meaning by position
- **Dict keys**: `grid[(0, 3)] = "start"` — (row, col) pairs
- **Set members**: `{(1,2), (3,4), (1,2)}` → dedup works
- **Multiple return values**: `return x, y` (Python returns a tuple implicitly)
- **Named tuples**: `collections.namedtuple` for readable field access
- **Constant data**: configuration, RGB colours, coordinates that *should not change*
- **Performance-critical code**: iteration over a tuple is marginally faster than over a list of the same size

Example 64: Immutability, Hashability and Memory

```python
import sys

% Tuple is immutable
t = (10, 20, 30)
try:
    t[0] = 99
except TypeError as e:
    print(e)   # 'tuple' object does not support item assignment

% But a mutable element inside can still change
t2 = (1, [2, 3], 4)
t2[1].append(99)         # list inside the tuple is mutable!
print(t2)                # (1, [2, 3, 99], 4)  -- tuple reference unchanged

% Hashable tuples can be dict keys
locations = {}
locations[(28.6, 77.2)] = "Delhi"
locations[(19.1, 72.8)] = "Mumbai"
print(locations[(28.6, 77.2)])   # Delhi

% Tuple with mutable element is NOT hashable
try:
    hash((1, [2, 3]))
except TypeError as e:
    print(e)   # unhashable type: 'list'

% Memory comparison
t3 = (1, 2, 3, 4, 5)
l3 = [1, 2, 3, 4, 5]
print(f"tuple: {sys.getsizeof(t3)} bytes")  # tuple: 80 bytes
print(f"list : {sys.getsizeof(l3)} bytes")  # list : 104 bytes

% Tuples as set members
points = {(0,0), (1,2), (3,4), (0,0)}   # duplicate removed
print(points)   # {(0, 0), (1, 2), (3, 4)}
```

2.3.2 Creation & Special Syntax

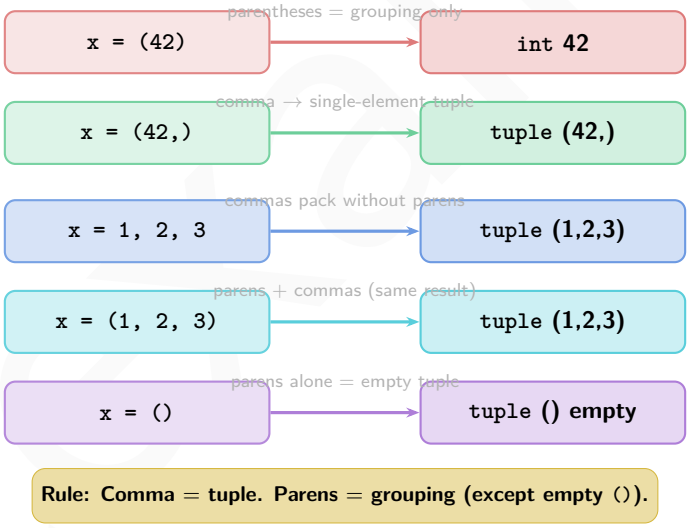
2.3.2.1 Literal Forms and the Comma Rule

Tuple Creation — The Comma Makes the Tuple, Not Parentheses

A common misconception: parentheses *do not* create tuples — the **comma does**. Parentheses are only for grouping or disambiguation.

Expression	Type	Value
`x = 1, 2, 3`	`tuple`	`(1, 2, 3)`
`x = (1, 2, 3)`	`tuple`	`(1, 2, 3)`
`x = (42)`	`int`	42 (just grouping!)
`x = (42,)`	`tuple`	`(42,)`
`x = ()`	`tuple`	`()` empty
`x = 42,`	`tuple`	`(42,)`

Single-element tuple: the trailing comma `(x,)` is **mandatory**. Without it, `(42)` is just an integer in parentheses.



All Creation Methods

- **Packing literal**: `t = 1, 2, 3` or `t = (1, 2, 3)`
- **Single-element**: `t = (x,)` or `t = x,` — trailing comma is *required*
- **Empty**: `t = ()` or `t = tuple()`
- **Constructor**: `tuple(iterable)` — converts any iterable
- **Repetition**: `(0,) * 5` → `(0, 0, 0, 0, 0)` (safe for immutables; same pitfall as lists for nested mutables)
- **Nested**: `t = ((1,2), (3,4), (5,6))` — tuple of tuples
- **Concatenation**: `(1,2) + (3,4)` → `(1,2,3,4)` (creates new tuple)

Example 65: Creating Tuples — All Forms

```
% Literal with parentheses
t1 = (1, 2, 3)
print(type(t1), t1)     # <class 'tuple'> (1, 2, 3)

% Parentheses are OPTIONAL (comma packs)
t2 = 1, 2, 3
print(type(t2), t2)     # <class 'tuple'> (1, 2, 3)
print(t1 == t2)         # True

% Single element: COMMA IS MANDATORY
a = (42,)
b = (42)                # this is just int 42!
print(type(a), a)       # <class 'tuple'> (42,)
print(type(b), b)       # <class 'int'>   42

% Trailing comma works without parens too
c = 42,
print(type(c), c)       # <class 'tuple'> (42,)

% Empty tuple
e = ()
print(type(e), len(e)) # <class 'tuple'> 0

% Constructor from iterables
t3 = tuple("GATE")              # ('G', 'A', 'T', 'E')
t4 = tuple(range(1, 6))         # (1, 2, 3, 4, 5)
t5 = tuple([10, 20, 30])        # from list
t6 = tuple({3, 1, 2})           # from set (order not guaranteed)
print(t3, t4, t5)

% Repetition
t7 = (0,) * 5
print(t7)               # (0, 0, 0, 0, 0)

% Concatenation creates NEW tuple
t8 = (1, 2) + (3, 4)
print(t8)               # (1, 2, 3, 4)

% Nested tuple
coords = ((0.0, 1.0), (3.5, 4.2), (-1.0, 2.0))
print(coords[1])        # (3.5, 4.2)
print(coords[1][0])     # 3.5
```

2.3.3 Packing & Unpacking

2.3.3.1 All Unpacking Forms

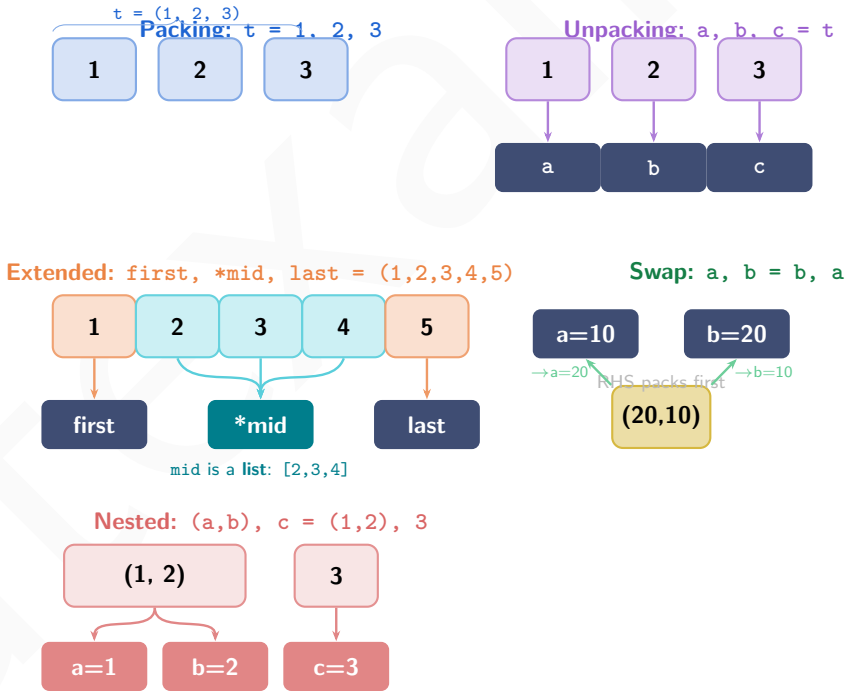
Tuple Packing and Unpacking

Packing: multiple values on the right of = are automatically gathered into a tuple.
Unpacking: a tuple (or any iterable) on the right of = is distributed across multiple names on the left. The number of names must match the number of elements (unless * is used).
Forms:

Form	Example
Basic unpacking	`a, b, c = (1, 2, 3)`
Extended (*)	`first, *rest = (1,2,3,4)`
Nested	`(a,b), c = (1,2), 3`
Swap	`a, b = b, a`
Discard	`_, y, _ = (1,2,3)`
Star in middle	`a, *b, c = (1,2,3,4,5)`
In for-loop	`for x,y in [(1,2),(3,4)]: ...`
Function args	`f(*t)` unpacks tuple as positional args

Rules:

- Only **one * variable** allowed per unpacking expression
- * always captures a **list** (not a tuple), even when unpacking a tuple
- Works for *any iterable* on the right, not just tuples



Extended Unpacking Rules

- **Only one** * per assignment: `a, *b, *c = t` is a `SyntaxError`
- *** always collects a list**: even when unpacking a tuple, the starred variable gets a `list` object
- **Zero or more elements**: `first, *rest = (1,)` gives `first=1`, `rest=[]`
- **Works on any iterable**: list, range, string, generator, etc.
- **Nested unpacking**: patterns can be nested to arbitrary depth

- **Underscore convention**: use _ for values you want to discard

Example 66: Basic and Extended Unpacking

```
% Basic
a, b, c = (10, 20, 30)
print(a, b, c)           # 10 20 30

% Too few / too many -- ValueError
try:
    x, y = (1, 2, 3)
except ValueError as e:
    print(e)   # too many values to unpack (expected 2)

% Extended unpacking
first, *rest = (1, 2, 3, 4, 5)
print(first)             # 1
print(rest)              # [2, 3, 4, 5]   <-- list!
print(type(rest))        # <class 'list'>

*init, last = (1, 2, 3, 4, 5)
print(init, last)        # [1, 2, 3, 4]  5

a, *mid, z = (1, 2, 3, 4, 5)
print(a, mid, z)         # 1  [2, 3, 4]  5

% Star captures zero elements too
first2, *empty, last2 = (1, 2)
print(first2, empty, last2)   # 1 [] 2

% Discarding values with _
_, y, _ = (1, 99, 3)
print(y)                 # 99

% Unpack in for-loop (very common!)
pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
for num, char in pairs:
    print(f"{num}: {char}")
% 1: a  /  2: b  /  3: c

% Works on strings (any iterable)
first_char, *rest_chars = "Python"
print(first_char)        # P
print(rest_chars)        # ['y', 't', 'h', 'o', 'n']
```

Example 67: Nested Unpacking

```
% Nested tuple unpacking
(a, b), c = (1, 2), 3
print(a, b, c)           # 1 2 3

% Unpack nested in for-loop
matrix = [(1, (2, 3)), (4, (5, 6)), (7, (8, 9))]
for x, (y, z) in matrix:
    print(f"x={x}, y={y}, z={z}")
% x=1, y=2, z=3
% x=4, y=5, z=6
% x=7, y=8, z=9

% Deep nesting
((a, b), (c, d)) = ((1, 2), (3, 4))
print(a, b, c, d)        # 1 2 3 4

% Mixed nesting with *
(first, *rest), last = (1, 2, 3), 4
print(first, rest, last) # 1 [2, 3] 4
```

```
% Practical: parse CSV-like records
records = [
    ("Alice", 25, ("Delhi", "IN")),
    ("Bob",   30, ("London", "UK")),
]
for name, age, (city, country) in records:
    print(f"{name}, {age}, {city}, {country}")
% Alice, 25, Delhi, IN
% Bob, 30, London, UK
```

Example 68: Swap and Function Unpacking

```
% Simultaneous swap -- RHS fully evaluated first
a, b = 10, 20
a, b = b, a                 # packs (20,10) then unpacks
print(a, b)                 # 20 10

% Three-way rotation
x, y, z = 1, 2, 3
x, y, z = y, z, x
print(x, y, z)              # 2 3 1

% Unpack as function arguments with *
def add3(x, y, z):
    return x + y + z

args = (3, 4, 5)
print(add3(*args))          # 12  (same as add3(3, 4, 5))

% Mixed: some explicit, some starred
def greet(name, greeting, punct):
    return f"{greeting}, {name}{punct}"

params = ("World", "!")
print(greet("Hello", *params))   # Hello, World!

% ** for dict unpacking into keyword args
def describe(name, age, city):
    return f"{name}, {age}, {city}"

info = {"age": 30, "city": "Delhi"}
print(describe("Alice", **info))  # Alice, 30, Delhi

% Function returning multiple values as tuple
def minmax(lst):
    return min(lst), max(lst)   # implicitly packs a tuple

low, high = minmax([3, 1, 4, 1, 5, 9])
print(low, high)            # 1 9

result = minmax([3, 1, 4])
print(type(result))         # <class 'tuple'>
```

2.3.4 Tuple Methods

2.3.4.1 `count()` and `index()`

Tuple Methods — Only Two

Tuples have exactly **two methods** (because they are immutable; no mutation methods exist):

Method	Returns	Behaviour
`count(x)`	`int`	Number of occurrences of `x`; returns 0 if absent
`index(x[, start, end])`	`int`	Index of first `x`; raises `ValueError` if absent

All other sequence operations are **built-in functions** (not methods):

Operation	Example	Note
Length	`len(t)`	$O(1)$
Membership	`x in t`	$O(n)$
Min / Max	`min(t)`, `max(t)`	elements must be comparable
Sum	`sum(t)`	numeric elements
Sort (new)	`sorted(t)`	returns a `list`
Reverse iterator	`reversed(t)`	returns iterator
Indexing	`t[i]`	$O(1)$
Slicing	`t[a:b:s]`	returns new tuple
Concatenation	`t1 + t2`	returns new tuple
Repetition	`t * n`	returns new tuple

Named Tuples — `collections.namedtuple`

Named tuples are a subclass of `tuple` that give each position a *name*, making records self-documenting without the overhead of a full class.

- Accessed by **name** (`p.x`) or **index** (`p[0]`)
- Still **immutable** (no assignment to fields)
- Usable anywhere a regular tuple is (hashing, dict keys, unpacking)
- `_asdict()` returns an `OrderedDict` of field values
- `_replace(**kwargs)` returns a *new* tuple with some fields changed
- Python 3.6+ **`typing.NamedTuple`**: cleaner class-based syntax with type hints

Example 69: `count()`, `index()` and All Sequence Operations

```
t = (3, 1, 4, 1, 5, 9, 2, 6, 1, 4)

% count: occurrences (no ValueError for missing)
print(t.count(1))          # 3
print(t.count(7))          # 0

% index: first occurrence
print(t.index(4))          # 2
print(t.index(4, 3))       # 9  (search from index 3)
try:
    t.index(99)
```

```
except ValueError as e:
    print(e)                # tuple.index(x): x not in tuple

% Built-in operations on tuples
print(len(t))               # 10
print(min(t), max(t))       # 1  9
print(sum(t))               # 36
print(1 in t)               # True
print(99 not in t)          # True

% sorted returns NEW LIST
print(sorted(t))            # [1, 1, 1, 2, 3, 4, 4, 5, 6, 9]
print(type(sorted(t)))      # <class 'list'>

% reversed returns iterator
print(list(reversed(t)))    # [4, 1, 6, 2, 9, 5, 1, 4, 1, 3]

% Slicing returns NEW TUPLE
print(t[2:6])               # (4, 1, 5, 9)
print(t[::-1])              # (4, 1, 6, 2, 9, 5, 1, 4, 1, 3)

% Concatenation and repetition
print((1, 2) + (3, 4))     # (1, 2, 3, 4)
print((0,) * 4)             # (0, 0, 0, 0)
```

Example 70: Named Tuples

```
from collections import namedtuple

% Define a named tuple type
Point = namedtuple('Point', ['x', 'y'])
Person = namedtuple('Person', 'name age city')    # space-separated too

% Create instances
p = Point(3.0, 4.0)
print(p)                # Point(x=3.0, y=4.0)
print(p.x, p.y)         # 3.0  4.0   (field access by name)
print(p[0], p[1])       # 3.0  4.0   (index access still works)

% Distance from origin
dist = (p.x**2 + p.y**2) ** 0.5
print(dist)             # 5.0

% Unpacking works as usual
x, y = p
print(x, y)             # 3.0  4.0

% _asdict: convert to dict
print(p._asdict())      # {'x': 3.0, 'y': 4.0}

% _replace: returns NEW tuple with changed fields
p2 = p._replace(x=10.0)
print(p2)               # Point(x=10.0, y=4.0)
print(p)                # Point(x=3.0, y=4.0)  original unchanged

% Hashable -- can be dict key or set member
locations = {Point(0,0): "origin", Point(1,0): "unit-x"}
print(locations[Point(0,0)])   # origin

% Python 3.6+: class-based syntax with type hints
from typing import NamedTuple

class Student(NamedTuple):
    name: str
    score: float
    rank: int = 0           # default value

s = Student("Alice", 95.5)
```

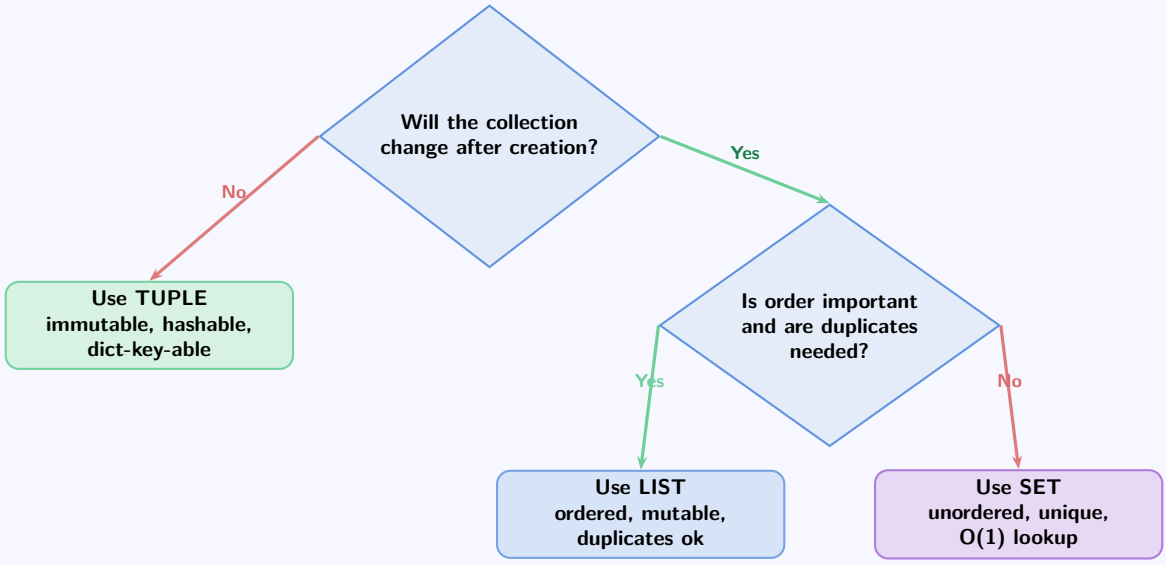
```
print(s)                        # Student(name='Alice', score=95.5, rank=0)
print(s.name, s.score)          # Alice  95.5

% Iterate namedtuple fields
for field, val in s._asdict().items():
    print(f"  {field}: {val}")
```

Example 71: Tuples as Dict Keys — Practical Uses

```
% 2D grid mapping using tuple keys
grid = {}
grid[(0, 0)] = "start"
grid[(3, 4)] = "end"
grid[(1, 2)] = "obstacle"

print(grid[(0, 0)])          # start
print((3, 4) in grid)        # True

% Memoisation with tuple key
memo = {}
def fib(n):
    if n <= 1:
        return n
    if (n,) not in memo:     # single-element tuple key
        memo[(n,)] = fib(n-1) + fib(n-2)
    return memo[(n,)]

% Multi-key grouping
from collections import defaultdict
sales = [
    ("Jan", "North", 500),
    ("Jan", "South", 300),
    ("Feb", "North", 700),
    ("Jan", "North", 200),
]
region_monthly = defaultdict(int)
for month, region, amount in sales:
    region_monthly[(month, region)] += amount   # tuple as compound key

for key, total in sorted(region_monthly.items()):
    print(f"{key}: {total}")
% ('Feb', 'North'): 700
% ('Jan', 'North'): 700
% ('Jan', 'South'): 300

% Sorting a list of tuples
data = [("Charlie", 78), ("Alice", 95), ("Bob", 78), ("Diana", 95)]
% Sort by score desc, then name asc
data.sort(key=lambda x: (-x[1], x[0]))
print(data)
% [('Alice', 95), ('Diana', 95), ('Bob', 78), ('Charlie', 78)]
```

2.3.4.2 Tuple vs List — When to Use Which

Tuple vs List Decision Guide



- Choose **tuples** for fixed structured records, coordinates, lookup keys, and static configuration constants to optimize performance and guard safety.
- Choose **lists** for collections of homogeneous elements that grow, shrink, dynamically reorder, or update frequently throughout execution.

2.4 Dictionaries

2.4.1 Dictionary Properties

2.4.1.1 Core Characteristics

Dictionary — Definition and Core Properties

A Python **dictionary** (`dict`) is an **ordered mapping** of **hashable keys** to **arbitrary values**. It is Python's primary associative data structure.

Property	Detail
Key–value pairs	Each entry is a *key* → *value* mapping
Keys: hashable	Must be immutable: `int`, `str`, `tuple` (of hashables), `bool`, `frozenset`
Values: any object	No restriction; can be lists, dicts, functions, `None`
Ordered (Python 3.7+)	Insertion order is preserved and guaranteed
No duplicate keys	Assigning to existing key *overwrites* the value
Mutable	Keys/values can be added, removed, updated

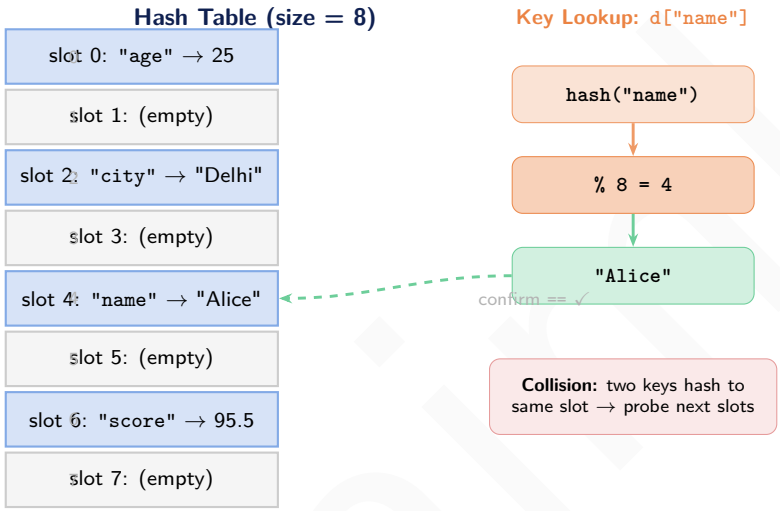
Hash Table Implementation

CPython implements `dict` as a **compact hash table**:

1. **Hash the key**: `hash(k)` produces an integer (e.g. `hash("name") = -8134...`)

2. **Compute slot**: `slot = hash(k) % table_size`
3. **Probe on collision**: if the slot is occupied by a *different* key, Python probes subsequent slots (open addressing with perturbation)
4. **Compare**: once the slot is reached, Python checks `k == stored_key` to confirm (hash equality does not guarantee key equality)

Resizing: when the load factor exceeds $\approx 2/3$, the table is resized (doubled) and all entries are rehashed — $O(n)$ occasionally, $O(1)$ amortised.



What Can Be a Dictionary Key?

- **Valid keys** (hashable): `int`, `float`, `str`, `bool`, `bytes`, `tuple` (of hashables), `frozenset`, `None`, custom objects with `__hash__`
- **Invalid keys** (unhashable): `list`, `dict`, `set` — raise `TypeError: unhashable type`
- **Equality determines uniqueness**: `True == 1` and `hash(True) == hash(1)`, so `{True: "a", 1: "b"}` has only *one* key (`True`) with value `"b"`
- **Float keys**: technically valid but *dangerous* due to floating-point imprecision; avoid using floats as keys

Example 72: Valid and Invalid Keys; True/1 Collision

```python
# Valid key types
d = {
    42:          "int key",
    3.14:        "float key",      # valid but risky
    "name":      "str key",
    True:        "bool key",
    (1, 2):      "tuple key",
    frozenset({1,2}): "frozenset key",
    None:        "None key",
}
print(d[42], d["name"], d[(1,2)])  # int key / str key / tuple key

# Invalid key: list raises TypeError
try:
    bad = {[1,2]: "value"}
except TypeError as e:
    print(e)   # unhashable type: 'list'

# True == 1 and hash(True) == hash(1): same key!
```

```python
d2 = {True: "first", 1: "second"}
print(d2)         # {True: 'second'}  -- only one entry; value overwritten
print(len(d2))    # 1

d3 = {False: "zero", 0: "also zero"}
print(d3)         # {False: 'also zero'}
print(len(d3))    # 1

# Ordered by insertion (Python 3.7+)
d4 = {"c": 3, "a": 1, "b": 2}
print(list(d4.keys()))    # ['c', 'a', 'b']  -- insertion order preserved
```

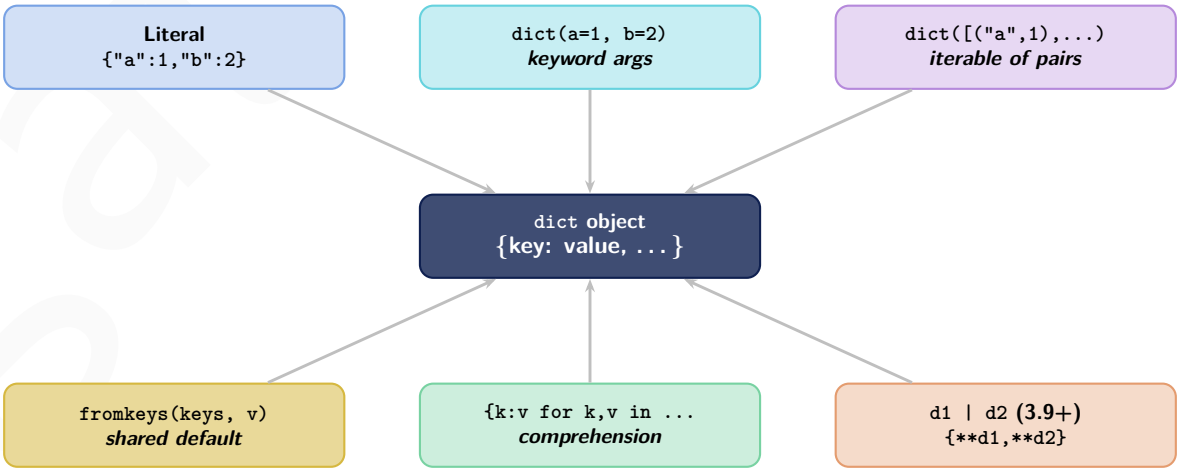
2.4.2 Dictionary Creation

2.4.2.1 All Creation Methods

Dictionary Creation Forms

Form	Example
Literal	`{"a": 1, "b": 2}`
Empty literal	`{}`
`dict()` keyword	`dict(a=1, b=2)`
`dict()` pairs	`dict([("a",1), ("b",2)])`
`dict()` from dict	`dict(existing_dict)`
`fromkeys()`	`dict.fromkeys(["a","b","c"], 0)`
Dict comprehension	`{k: v for k,v in pairs}`
Merge | (3.9+)	`d1 | d2`
Unpack **	`{**d1, **d2}`

fromkeys pitfall: `dict.fromkeys(keys, [])` shares the *same* list object for all keys — same pitfall as `[[]] * n` in lists.



Example 73: All Creation Methods

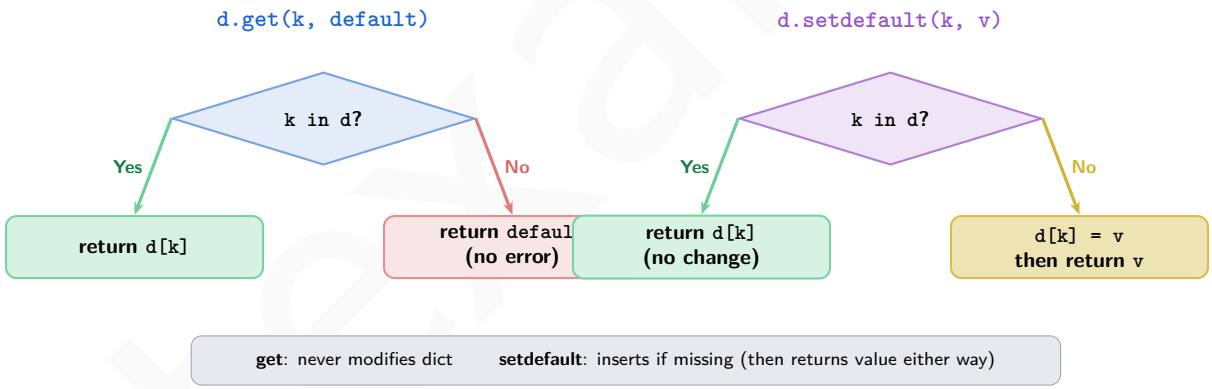
```python
# Literal
d1 = {"name": "Alice", "age": 25, "city": "Delhi"}

# dict() with keyword arguments (keys must be valid identifiers)
d2 = dict(name="Alice", age=25, city="Delhi")
print(d1 == d2)        # True

# dict() from list of (key, value) pairs
pairs = [("x", 10), ("y", 20), ("z", 30)]
d3 = dict(pairs)
print(d3)              # {'x': 10, 'y': 20, 'z': 30}

# dict() from zip
keys   = ["a", "b", "c"]
values = [1, 2, 3]
d4 = dict(zip(keys, values))
print(d4)              # {'a': 1, 'b': 2, 'c': 3}

# fromkeys: all keys get the SAME default
d5 = dict.fromkeys(["x", "y", "z"], 0)
print(d5)              # {'x': 0, 'y': 0, 'z': 0}

# fromkeys PITFALL with mutable default
d6 = dict.fromkeys(["a", "b", "c"], [])   # shared list!
d6["a"].append(1)
print(d6)      # {'a': [1], 'b': [1], 'c': [1]}  ALL changed!

# Correct: use comprehension for independent defaults
d7 = {k: [] for k in ["a", "b", "c"]}
d7["a"].append(1)
print(d7)      # {'a': [1], 'b': [], 'c': []}  only 'a' changed

# Merge two dicts (Python 3.9+)
base    = {"a": 1, "b": 2}
updates = {"b": 99, "c": 3}
merged  = base | updates
print(merged)   # {'a': 1, 'b': 99, 'c': 3}  (right dict wins on conflict)

# Unpack merge (all Python 3 versions)
merged2 = {**base, **updates}
print(merged2)  # {'a': 1, 'b': 99, 'c': 3}
```

2.4.3 Access & Mutation

2.4.3.1 Reading, Writing, Deleting

Accessing and Modifying Dictionary Entries

Operation	Syntax	Key Absent Behaviour
Get value	`d[k]`	`KeyError`
Get with default	`d.get(k, default)`	returns `default` (def: `None`)
Set / update	`d[k] = v`	inserts if new; overwrites if exists
Delete	`del d[k]`	`KeyError` if absent
Pop	`d.pop(k)`	`KeyError` if absent
Pop with default	`d.pop(k, default)`	returns `default`; no error
Pop last item	`d.popitem()`	removes & returns last `(k,v)` pair
Set-if-missing	`d.setdefault(k, v)`	inserts `v` if `k` absent; always returns value
Merge in-place	`d.update(other)`	adds/overwrites from `other`
Merge new dict	`d | other` (3.9+)	new dict; right side wins
Update in-place	`d |= other` (3.9+)	in-place version of `update`
Check key exists	`k in d`	$O(1)$; `not in` also works



get vs setdefault vs defaultdict

- **`d.get(k, default)`**: read-only safe access; does not modify dict
- **`d.setdefault(k, default)`**: ensures key exists with default; *inserts* on first miss; idiomatic for building grouped dicts
- **`collections.defaultdict(factory)`**: auto-inserts `factory()` for any missing key; cleaner than `setdefault` for aggregation patterns
- **Never** use `d[k]` when key may be absent unless you handle `KeyError` explicitly
- **`d.pop(k, None)`**: safe delete — never raises `KeyError`

Example 74: get, setdefault, pop, update — Full Examples

```
1 d = {"name": "Alice", "age": 25}
2
```

```python
# d[k] vs d.get()
print(d["name"])              # Alice
try:
    print(d["score"])         # KeyError!
except KeyError as e:
    print(f"KeyError: {e}")

print(d.get("score"))         # None  (no error)
print(d.get("score", 0))      # 0     (custom default)
print(d.get("name",  "?"))    # Alice (key exists, default ignored)

# setdefault: insert-then-return
d.setdefault("score", 100)    # inserts "score": 100
print(d["score"])             # 100
d.setdefault("score", 200)    # key already exists: no change
print(d["score"])             # 100  (still 100!)

# Grouping with setdefault
words = ["apple","ant","banana","blueberry","avocado","cherry"]
grouped = {}
for word in words:
    grouped.setdefault(word[0], []).append(word)
print(grouped)
# {'a': ['apple','ant','avocado'], 'b': ['banana','blueberry'], 'c': ['cherry']}

# pop
removed = d.pop("age")        # removes and returns
print(removed)                # 25
print("age" in d)             # False

d.pop("nonexistent", "safe")  # no KeyError with default
# d.pop("nonexistent")        # would raise KeyError!

# popitem: removes last inserted item (Python 3.7+ guaranteed LIFO)
d2 = {"x": 1, "y": 2, "z": 3}
print(d2.popitem())           # ('z', 3)
print(d2)                     # {'x': 1, 'y': 2}

# update: merge in-place
base = {"a": 1, "b": 2}
base.update({"b": 99, "c": 3})   # b overwritten, c added
print(base)   # {'a': 1, 'b': 99, 'c': 3}

base.update(d=4, e=5)            # keyword form
print(base)   # {'a': 1, 'b': 99, 'c': 3, 'd': 4, 'e': 5}

# |= in-place merge (Python 3.9+)
d3 = {"x": 1}
d3 |= {"y": 2, "z": 3}
print(d3)   # {'x': 1, 'y': 2, 'z': 3}
```

Example 75: defaultdict — Cleaner Aggregation

```python
from collections import defaultdict

# defaultdict(list): auto-creates [] for missing keys
grouped = defaultdict(list)
words = ["apple","ant","banana","blueberry","avocado","cherry"]
for word in words:
    grouped[word[0]].append(word)   # no setdefault needed!
print(dict(grouped))
# {'a': ['apple','ant','avocado'], 'b': ['banana','blueberry'], 'c': ['cherry']}

# defaultdict(int): word frequency counter
sentence = "the quick brown fox jumps over the lazy fox"
freq = defaultdict(int)
for word in sentence.split():
    freq[word] += 1             # missing key auto-initialised to 0
```

```
print(dict(freq))
# {'the': 2, 'quick': 1, 'brown': 1, 'fox': 2, ...}

# defaultdict(set): group into sets (dedup)
pairs = [("A","x"),("B","y"),("A","z"),("B","y"),("A","x")]
mapping = defaultdict(set)
for k, v in pairs:
    mapping[k].add(v)
print(dict(mapping))   # {'A': {'x', 'z'}, 'B': {'y'}}
```

2.4.4 Views & Iteration

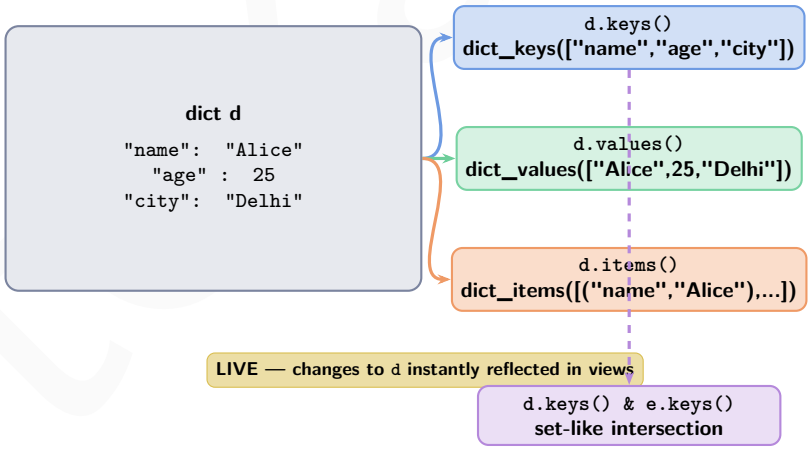
2.4.4.1 Live Views: keys(), values(), items()

Dictionary Views — Live, Dynamic Objects

The three view objects returned by `d.keys()`, `d.values()`, `d.items()` are **live views**: they *reflect changes* to the dictionary in real time without creating copies.

View	Type	Elements	Set-like?
`d.keys()`	`dict_keys`	all keys	Yes (if hashable)
`d.values()`	`dict_values`	all values	No
`d.items()`	`dict_items`	`(key,value)` tuples	Yes (if values hashable)

Set operations on views (keys and items): `d.keys() & other.keys()` (intersection), `d.keys() | other.keys()` (union), `d.keys() - other.keys()` (difference).



Iteration Patterns and Safe Modification

- **Iterate keys**: `for k in d:` (same as `for k in d.keys()`)
- **Iterate values**: `for v in d.values():`
- **Iterate key-value**: `for k, v in d.items():` (most common)
- **Never modify a dict while iterating over it** — raises `RuntimeError: dictionary changed size during iteration`
- **Safe delete during iteration**: iterate over `list(d.keys())` (a copy) and delete from the original
- `k in d` tests key membership in $O(1)$; `v in d.values()` is $O(n)$

Example 76: Iterating Over Views

```
d = {"name": "Alice", "age": 25, "city": "Delhi", "score": 95.5}

% Iterate keys (default)
for k in d:
    print(k, end=" ")      # name age city score

% Iterate values
for v in d.values():
    print(v, end=" ")      # Alice 25 Delhi 95.5

% Iterate items (most useful)
for key, val in d.items():
    print(f"{key}: {val}")

% Views are LIVE
kv = d.keys()
print(list(kv))            # ['name', 'age', 'city', 'score']
d["rank"] = 1
print(list(kv))            # ['name', 'age', 'city', 'score', 'rank']  updated!

% Set operations on keys
d1 = {"a": 1, "b": 2, "c": 3}
d2 = {"b": 20, "c": 30, "d": 40}
print(d1.keys() & d2.keys())   # {'b', 'c'}   intersection
print(d1.keys() | d2.keys())   # {'a','b','c','d'}  union
print(d1.keys() - d2.keys())   # {'a'}  difference

% Membership: O(1) for keys, O(n) for values
print("name" in d)            # True   O(1)
print("Alice" in d.values())  # True   O(n) -- linear scan!

% Safe deletion during iteration
d3 = {"a": 1, "b": -2, "c": 3, "d": -4, "e": 5}
for k in list(d3.keys()):      # copy the keys!
    if d3[k] < 0:
        del d3[k]
print(d3)   # {'a': 1, 'c': 3, 'e': 5}
```

Example 77: Useful Iteration Patterns

```
inventory = {"apple": 50, "banana": 0, "cherry": 30, "date": 0}

# Filter: items with non-zero stock
in_stock = {k: v for k, v in inventory.items() if v > 0}
print(in_stock)   # {'apple': 50, 'cherry': 30}

# Invert a dict (value -> key)
original = {"a": 1, "b": 2, "c": 3}
inverted = {v: k for k, v in original.items()}
print(inverted)   # {1: 'a', 2: 'b', 3: 'c'}

# Sort by value (returns list of tuples)
scores = {"Alice": 85, "Bob": 92, "Charlie": 78, "Diana": 95}
sorted_scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)
print(sorted_scores)
# [('Diana', 95), ('Bob', 92), ('Alice', 85), ('Charlie', 78)]

# Sorted dict (preserves insertion order after sort in 3.7+)
sorted_dict = dict(sorted(scores.items(), key=lambda x: x[1]))
print(sorted_dict)

# Enumerate with dict
for i, (k, v) in enumerate(scores.items(), 1):
    print(f"{i}. {k}: {v}")

# Merge and aggregate
```

```python
monthly = [
    {"region": "North", "sales": 500},
    {"region": "South", "sales": 300},
    {"region": "North", "sales": 200},
]
total = {}
for entry in monthly:
    total[entry["region"]] = total.get(entry["region"], 0) + entry["sales"]
print(total)   # {'North': 700, 'South': 300}
```

2.4.5 Nested Dictionaries

2.4.5.1 Access, Mutation, and the Sharing Trap

Nested Dictionaries — Structure and Access

A **nested dictionary** is a dictionary whose *values* are themselves dictionaries (or other containers). This models hierarchical data (JSON, configs, databases).

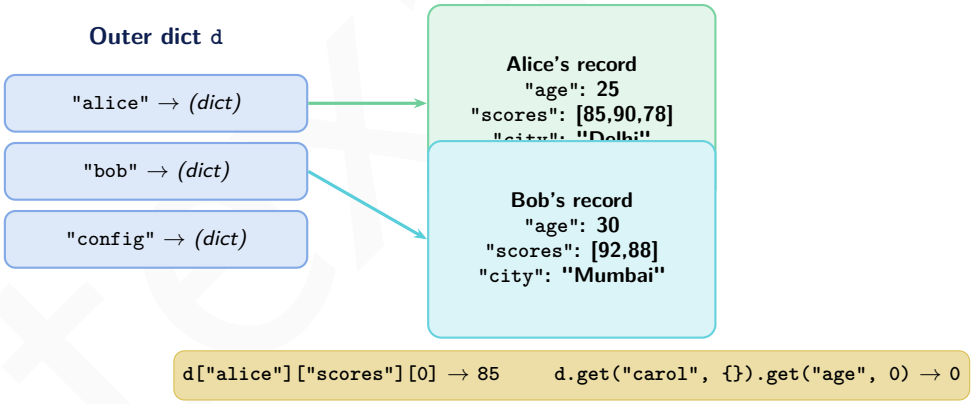
Accessing nested values:

`d['outer']['inner']` ≡ `temp = d['outer']; temp['inner']`

Safe access for deep nesting:

`d.get('outer', {}).get('inner', default)`

Mutation trap: when a mutable object (list, dict) is a *value* in two different dictionaries (because one was assigned from the other without a deep copy), modifying through one dict modifies both.



Nested Dict Patterns and Tools

- **Safe deep access**: `d.get("k1", {}).get("k2", default)` chains safely without `KeyError`
- `collections.ChainMap`: overlay multiple dicts without copying
- **Deep copy for independence**: `copy.deepcopy(d)` when nested mutable values must be independent
- **Flattening**: use recursive function or `pandas.json_normalize`
- **JSON roundtrip**: `json.dumps(d)` ↔ `json.loads(s)` — nested dicts map directly to JSON objects
- **Avoid** key chains longer than 3 levels — prefer a class or `dataclass` instead

Example 78: Nested Dict — Access and Safe Traversal

```python
# Building a nested dict
students = {
    "alice": {"age": 25, "scores": [85, 90, 78], "city": "Delhi"},
    "bob":   {"age": 30, "scores": [92, 88],      "city": "Mumbai"},
}

# Access: chained []
print(students["alice"]["age"])          # 25
print(students["alice"]["scores"][0])    # 85

# Modify nested value
students["alice"]["scores"].append(95)
print(students["alice"]["scores"])       # [85, 90, 78, 95]

# Add new nested key
students["alice"]["rank"] = 1
print(students["alice"])
# {'age': 25, 'scores': [85, 90, 78, 95], 'city': 'Delhi', 'rank': 1}

# Safe access: missing key at any level
print(students.get("carol", {}).get("age", 0))   # 0  (no KeyError)
print(students.get("alice", {}).get("phone", "N/A"))  # N/A

# KeyError example
try:
    print(students["carol"]["age"])  # KeyError: 'carol'
except KeyError as e:
    print(f"KeyError: {e}")

% Iterate nested
for name, info in students.items():
    avg = sum(info["scores"]) / len(info["scores"])
    print(f"{name.title()}: avg = {avg:.1f}, city = {info['city']}")
# Alice: avg = 87.0, city = Delhi
# Bob  : avg = 90.0, city = Mumbai
```

Example 79: Nested Dict Mutation Trap — Shared References

```python
import copy

# TRAP: shallow copy shares nested dicts
original = {
    "alice": {"scores": [85, 90]},
    "bob":   {"scores": [92, 88]},
}
shallow = original.copy()   # shallow copy

# Outer dict: independent
shallow["carol"] = {"scores": [70]}
print("carol" in original)  # False   (outer is independent)

# Inner dict: SHARED
shallow["alice"]["scores"].append(99)
print(original["alice"]["scores"])   # [85, 90, 99]  MODIFIED!

# CORRECT: deep copy
original = {
    "alice": {"scores": [85, 90]},
    "bob":   {"scores": [92, 88]},
}
deep = copy.deepcopy(original)
deep["alice"]["scores"].append(99)
print(original["alice"]["scores"])   # [85, 90]  unchanged

# TRAP: fromkeys with mutable default (same issue)
d = dict.fromkeys(["alice", "bob"], {"scores": []})
```

```python
d["alice"]["scores"].append(100)
print(d)  # {'alice': {'scores': [100]}, 'bob': {'scores': [100]}} -- Both mutated!
```

2.5 Sets & Frozensets

2.5.1 Properties

2.5.1.1 Core Characteristics

Set and Frozenset — Definitions

A Python **set** is an **unordered collection of unique, hashable elements**. It models the mathematical notion of a set.

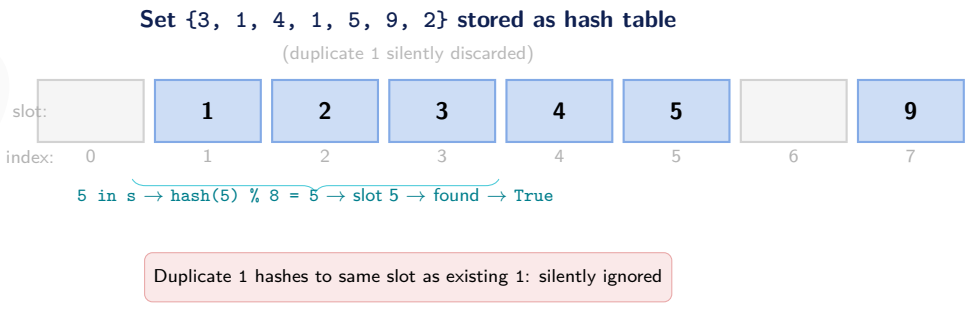
Property	set	frozenset	list (contrast)
Ordered	No	No	Yes
Unique elements	Yes	Yes	No
Mutable	**Yes**	**No**	Yes
Hashable	No	**Yes**	No
Dict key / set member	No	**Yes**	No
Membership `in`	$O(1)$ avg	$O(1)$ avg	$O(n)$
Elements must be	hashable	hashable	any

Frozenset is the *immutable* version of `set`. Because it is immutable it is hashable and can be used as a dict key or as an element of another set.

Hash-Table Implementation

Like `dict`, both `set` and `frozenset` are implemented as **hash tables**, which gives:

- $O(1)$ **average time complexity** for `add`, `remove`, `discard`, and `in` (membership testing).
- $O(n)$ **worst-case time complexity** (driven by systematic hash collisions, though exceptionally rare with robust internal hash distributions).
- No inherent ordering: iterating a set yields elements in an *arbitrary* (but deterministic within a single process lifecycle) execution order.
- Elements must be explicitly **hashable**: attempting to insert a mutable container like a `list` instantly triggers a `TypeError: unhashable type: 'list'`.



Set vs Frozenset Use Cases

- **set**: Data deduplication, rapid empirical membership evaluation, implementation of discrete mathematical set algebra operations on dynamic mutations.
- **frozenset**: Keys within a `dict` schema requiring structural compounding, multi-element sets of intersecting criteria (*set of sets*), unalterable system wide global flags/registries, and secure shared-state thread operations.
- **Algorithmic Selection Rule**: Choose a `set` over a linear sequence (`list`) whenever processing scale compounds alongside dense, repetitive membership validation iterations ($O(1)$ constant vs. $O(n)$ linear sweep cost).
- **Truthiness evaluations**: An uninstantiated or empty structural reference `set()` maps cleanly to boolean `False`; any non-empty variant resolves explicitly to `True`.

Example 80: Set Properties — Uniqueness, Ordering, Hashability

```python
# Duplicates automatically removed
s = {3, 1, 4, 1, 5, 9, 2, 6, 5}
print(s)          # {1, 2, 3, 4, 5, 6, 9}  (order not guaranteed)
print(len(s))     # 7  (not 9 -- duplicates gone)

# No ordering: index access raises TypeError
try:
    print(s[0])
except TypeError as e:
    print(e)      # 'set' object is not subscriptable

# O(1) membership test (vs O(n) for list)
print(5 in s)     # True
print(7 in s)     # False

# Elements must be hashable
try:
    bad = {[1, 2], [3, 4]}
except TypeError as e:
    print(e)      # unhashable type: 'list'

# frozenset is hashable -- can be dict key or set element
fs = frozenset({1, 2, 3})
d = {fs: "frozen group"}
print(d[frozenset({1, 2, 3})])    # frozen group

# set of frozensets
groups = {frozenset({1,2}), frozenset({3,4}), frozenset({1,2})}
print(groups)     # {frozenset({1, 2}), frozenset({3, 4})}  (deduped)

# Truthiness
print(bool(set()))           # False
print(bool({0}))             # True  (non-empty, even if element is 0)
```

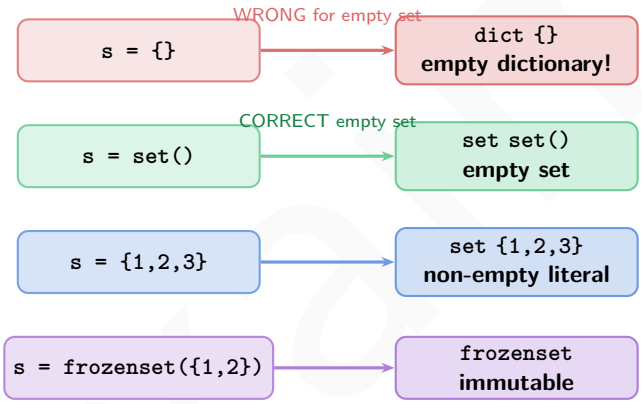
2.5.2 Set Creation

2.5.2.1 Literals and Constructors

Creating Sets and Frozensets

Expression	Type	Note
`{1, 2, 3}`	`set`	Literal instantiation (not empty dict!)
`{}`	`dict`	**Empty dictionary syntax reference!**
`set()`	`set`	Empty set functional generator
`set(iterable)`	`set`	Constructor parsing any iterable sequence
`frozenset(iterable)`	`frozenset`	Immutable immutable generator variant
`{x for x in ... if}`	`set`	High-speed declarative set comprehension

Critical Architectural Warning: Standard structural compiler mapping rules route explicit {} declarations directly to an unallocated *empty dict*. An explicit target conversion sequence must utilize the explicit constructor function `set()`.



Example 81: All Creation Methods

```python
# Literal: {1, 2, 3}
s1 = {3, 1, 4, 1, 5}
print(s1, type(s1))    # {1, 3, 4, 5}  <class 'set'>

# EMPTY SET: must use set(), not {}
empty_set  = set()
empty_dict = {}
print(type(empty_set))   # <class 'set'>
print(type(empty_dict))  # <class 'dict'>  <- common mistake!

# From iterables
s2 = set("abracadabra")        # {'a','b','c','d','r'}
s3 = set([1, 2, 2, 3, 3, 3]) # {1, 2, 3}
s4 = set(range(0, 10, 2))      # {0, 2, 4, 6, 8}
s5 = set((10, 20, 10, 30))     # {10, 20, 30}

# Set comprehension
squares = {x**2 for x in range(1, 6)}
print(squares)   # {1, 4, 9, 16, 25}

evens = {x for x in range(20) if x % 2 == 0}
print(evens)     # {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}

# frozenset
fs1 = frozenset({1, 2, 3})
fs2 = frozenset("python")
print(fs1)       # frozenset({1, 2, 3})
```

```python
print(fs2)        # frozenset({'p','y','t','h','o','n'})

# frozenset is hashable: use as dict key
edges = {frozenset({"A","B"}): 5, frozenset({"B","C"}): 3}
print(edges[frozenset({"A","B"})])   # 5

# Deduplication idiom: list -> set -> list
lst = [3,1,4,1,5,9,2,6,5,3,5]
unique = list(set(lst))        # order NOT preserved
print(sorted(unique))          # [1, 2, 3, 4, 5, 6, 9]

# Preserve order while deduping (Python 3.7+ dict keeps order)
seen = {}
unique_ordered = list(dict.fromkeys(lst))
print(unique_ordered)   # [3, 1, 4, 5, 9, 2, 6]  insertion order preserved
```

2.5.3 Set Operations

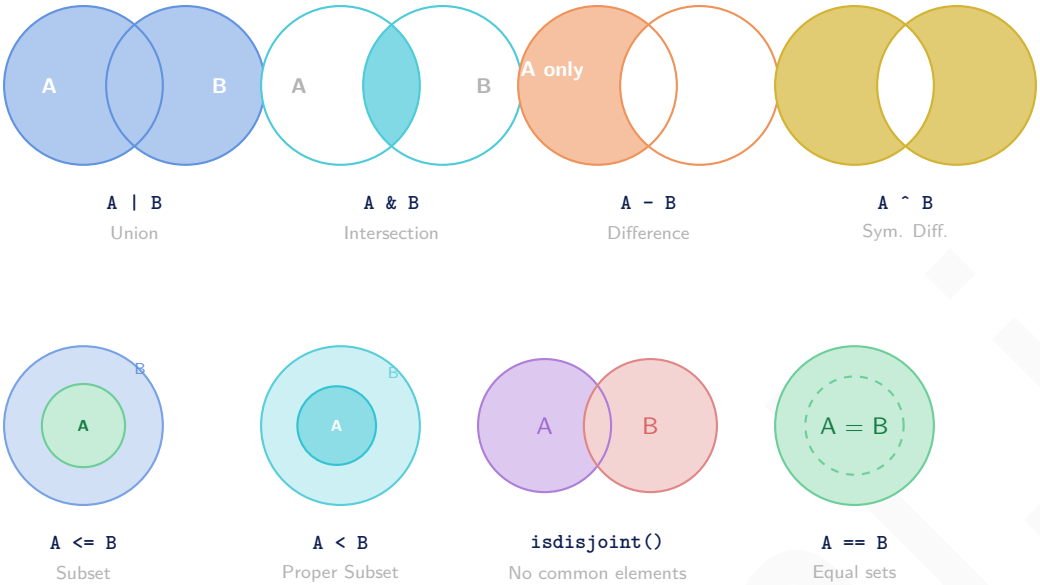
2.5.3.1 Mathematical Set Operators

Set Algebra — Operators and Method Equivalents

Python provides both highly structured **operator syntax** (restricting interaction evaluation properties explicitly between strict matching set objects) and programmatic **method syntax** (accepting any arbitrary *iterable* as an evaluation argument):

Operation	Operator	Method	In-place op	In-place method
Union	`A | B`	`A.union(B)`	`A |= B`	`A.update(B)`
Intersection	`A & B`	`A.intersection(B)`	`A &= B`	`A.intersection_update(B)`
Difference	`A - B`	`A.difference(B)`	`A -= B`	`A.difference_update(B)`
Symmetric difference	`A ^ B`	`A.symmetric_difference(B)`	`A ^= B`	`A.symmetric_difference_update(B)`
Subset	`A <= B`	`A.issubset(B)`	—	—
Proper subset	`A < B`	—	—	—
Superset	`A >= B`	`A.issuperset(B)`	—	—
Proper superset	`A > B`	—	—	—
Disjoint (no overlap)	—	`A.isdisjoint(B)`	—	—

Type Constraint Difference: `A | B` enforces compilation constraints requiring clean type uniformity across all operators. Conversely, invocation properties like `A.union(B)` implicitly execute cast operations across any operational container target sequence passed within parameter scopes.



Operator vs Method — Key Difference

- **Operators** (`|`, `&`, `-`, `^`): Enforce strict verification rules requiring both operational parameters to conform explicitly as `set` or `frozenset` instances.
- **Methods** (`.union()`, `.intersection()`, ...): Provide extensive execution latitude by unpacking any sequential stream or iterable entity passed during operational lifecycle execution.
- Modular expansion strategies enable execution calls across several arguments simultaneously, mapping the conversion equation `A.union(B, C, D)` identically to `A | B | C | D`.
- **In-place operations** execute memory conservation parameters by directly restructuring active state definitions over target addresses; these structural shifts fail under immutable `frozenset` boundaries.
- Type system consistency dictates that calls dispatched against structural variable instances preserve parent instance types exactly upon execution output generation.

Example 82: All Set Algebra Operations

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Union: all elements from both
print(A | B)                    # {1, 2, 3, 4, 5, 6, 7, 8}
print(A.union(B))               # same
print(A.union([4,5,6,7,8]))     # method accepts any iterable!
% print(A | [4,5,6])            # TypeError: operator needs set

# Intersection: only common elements
print(A & B)                    # {4, 5}
print(A.intersection(B))        # {4, 5}

# Difference: in A but not B
print(A - B)                    # {1, 2, 3}
print(B - A)                    # {6, 7, 8}  (not symmetric!)

# Symmetric difference: in exactly one of A, B
print(A ^ B)                    # {1, 2, 3, 6, 7, 8}
print(A.symmetric_difference(B))  # same

# Multiple sets with methods
C = {5, 6, 9}
print(A.union(B, C))            # {1,2,3,4,5,6,7,8,9}
print(A.intersection(B, C))     # {5}   common to all three
```

```
# In-place operations
X = {1, 2, 3, 4}
X |= {4, 5, 6}                  # X = X | {4,5,6}
print(X)                        # {1, 2, 3, 4, 5, 6}

Y = {1, 2, 3, 4, 5}
Y &= {3, 4, 5, 6}
print(Y)                        # {3, 4, 5}

Z = {1, 2, 3, 4, 5}
Z -= {3, 4}
print(Z)                        # {1, 2, 5}

W = {1, 2, 3, 4}
W ^= {3, 4, 5, 6}
print(W)                        # {1, 2, 5, 6}
```

Example 83: Subset, Superset, Disjoint Checks

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}
C = {4, 5, 6}
D = {1, 2, 3}    # same as A

# Subset: A <= B  (A is subset of B)
print(A <= B)               # True   (all of A is in B)
print(A.issubset(B))        # True
print(B <= A)               # False  (B has elements not in A)

# Proper subset: A < B  (subset AND not equal)
print(A < B)                # True   (A subset of B, A != B)
print(A < D)                # False  (A == D, so NOT proper subset)
print(A <= D)               # True   (A == D, still a subset)

# Superset: B >= A
print(B >= A)               # True
print(B.issuperset(A))      # True
print(B > A)                # True   (proper superset)
print(D > A)                # False  (D == A)

# Disjoint: no common elements
print(A.isdisjoint(C))      # True   (A={1,2,3}, C={4,5,6})
print(A.isdisjoint(B))      # False  (share 1,2,3)
print(A.isdisjoint([7,8])) # True   (method accepts any iterable)

# Equality
print(A == D)               # True
print(A == B)               # False

# Practical validation pattern: verify structural dictionary signatures
required = {"name", "age", "email"}
form_data = {"name": "Alice", "age": 25, "email": "a@b.com", "city": "Delhi"}
print(required <= form_data.keys())  # True (all required keys present)

missing = required - form_data.keys()
print(missing)   # set()  (no missing keys)
```

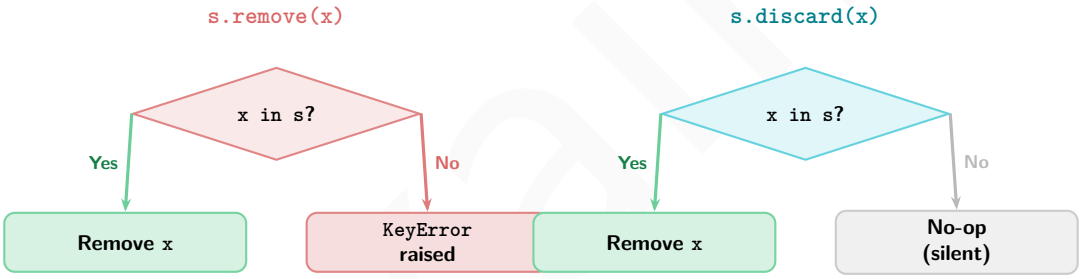
2.5.3.2 Mutation Methods

Set Mutation Methods

Method	Behaviour	Key Absent?
`add(x)`	Inserts element `x`; identity no-op if mapping entry overlaps	—
`remove(x)`	Destroys element `x`; explicitly triggers structural fault error	`KeyError`
`discard(x)`	Destroys element `x`; **safely bypasses** structural faults	silent no-op
`pop()`	Extracts and targets arbitrary current structural index storage data	`KeyError`
`clear()`	Wipes out current allocated storage space down to standard `set()`	—

Defensive Choice: remove vs discard:

- Leverage **`remove(x)`** when missing query components reveal logic breaking validation states across system runtimes.
- Leverage **`discard(x)`** when structural mapping configurations do not promise structural membership continuity during execution passes.



Example 84: add, remove, discard, pop, clear

```python
s = {1, 2, 3, 4, 5}

# add: no-op if already present
s.add(6)
print(s)                # {1, 2, 3, 4, 5, 6}
s.add(3)                # already in set: no change, no error
print(s)                # {1, 2, 3, 4, 5, 6}  unchanged

# remove: raises KeyError if missing
s.remove(6)
print(s)                # {1, 2, 3, 4, 5}
try:
    s.remove(99)
except KeyError as e:
    print(f"KeyError: {e}")    # KeyError: 99

# discard: silent if missing
s.discard(5)            # 5 was in set: removed
print(s)                # {1, 2, 3, 4}
s.discard(99)           # 99 not in set: no error
print(s)                # {1, 2, 3, 4}  unchanged

# pop: removes and returns arbitrary element
s2 = {10, 20, 30}
val = s2.pop()
print(val)              # some element (arbitrary: 10 or 20 or 30)
print(s2)               # remaining 2 elements
```

```python

# pop on empty set
try:
    empty = set()
    empty.pop()
except KeyError as e:
    print(e)    # 'pop from an empty set'

# clear: empty the set
s3 = {1, 2, 3}
s3.clear()
print(s3)              # set()
print(len(s3))         # 0

# In-place update methods (same as |=, &=, -=, ^=)
A = {1, 2, 3, 4}
A.update([5, 6, 7])                    # A |= {5,6,7}
print(A)               # {1, 2, 3, 4, 5, 6, 7}

A.intersection_update([2, 4, 6, 8])    # A &= {2,4,6,8}
print(A)               # {2, 4, 6}

A.difference_update([4])               # A -= {4}
print(A)               # {2, 6}

A.symmetric_difference_update([2,3,4])  # A ^= {2,3,4}
print(A)               # {3, 4, 6}
```

2.5.3.3 Frozenset Details and Practical Set Patterns

Frozenset — Extra Properties

- **Complete Evaluation Symmetry**: Every non-mutating query pipeline supported by standard mutable instances operates cleanly on immutable variants: `in`, `|`, `&`, `-`, `^`, `issubset`, `issuperset`, `isdisjoint`, data-length metrics, and sequence loops.
- **Total Immutability Protection**: Mutation gateways such as `add`, `remove`, `discard`, `pop`, `clear`, and `update` are completely omitted from the interface definition.
- **Heterogeneous Computation Rules**: Intersection evaluation patterns yield types derived from the primary operand sequence: `frozenset & set` generates a `frozenset`; reversing evaluation sequences shifts output allocation cleanly to standard `set` configurations.
- **Hash Consistency Verification**: Evaluating structural inputs via `hash(frozenset({1,2,3}))` processes securely; checking matching state definitions via `frozenset({1,2}) == {1,2}` tracks cleanly to mathematical evaluation criteria.

Example 85: Frozenset Operations

```python
fs = frozenset({1, 2, 3, 4, 5})
s  = {3, 4, 5, 6, 7}

# All read operations work
print(3 in fs)            # True
print(fs & s)             # frozenset({3, 4, 5})
print(fs | s)             # frozenset({1, 2, 3, 4, 5, 6, 7})
print(fs - s)             # frozenset({1, 2})
print(fs.issubset({1,2,3,4,5,6}))  # True

# Hashable: dict key
graph_edges = {
    frozenset({"A","B"}): 5,
    frozenset({"B","C"}): 3,
    frozenset({"A","C"}): 7,
```

```
}
print(graph_edges[frozenset({"A","C"})])   # 7

# Hashable: element of set
families = {
    frozenset({"Alice","Bob"}),
    frozenset({"Charlie","Diana"}),
    frozenset({"Alice","Bob"}),    # duplicate -- removed
}
print(len(families))   # 2

# Mutation raises AttributeError
try:
    fs.add(6)
except AttributeError as e:
    print(e)   # 'frozenset' object has no attribute 'add'

# Mixed operations: frozenset op set -> frozenset; set op frozenset -> set
print(type(fs & s))        # <class 'frozenset'>
print(type(s & fs))        # <class 'set'>

# Equality across types
print(frozenset({1,2,3}) == {1,2,3})   # True
```

Example 86: Practical Set Patterns for Computational Evaluation

```
# 1. High-Speed Linear Array Deduplication
data = [1,2,3,2,4,1,5,3,6]
unique = list(set(data))          # order not preserved
print(sorted(unique))             # [1, 2, 3, 4, 5, 6]

# 2. Performance Profiling: Linear Array Sweep vs. Constant Hash Lookup
import time
large_list = list(range(1_000_000))
large_set  = set(range(1_000_000))

t0 = time.perf_counter()
_ = 999_999 in large_list
print(f"list: {time.perf_counter()-t0:.4f}s")   # ~0.01s (O(n) dependency loop)

t0 = time.perf_counter()
_ = 999_999 in large_set
print(f"set : {time.perf_counter()-t0:.6f}s")   # ~0.0000001s (O(1) constant index look)

# 3. Dissecting Overlaps and Set Variances
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]
common   = set(list1) & set(list2)   # {3, 4, 5}
only_l1  = set(list1) - set(list2)   # {1, 2}
only_l2  = set(list2) - set(list1)   # {6, 7}
all_uniq = set(list1) ^ set(list2)   # {1, 2, 6, 7}

# 4. Identity Evaluation Mechanics (Character Frequency Invariant Check)
def is_same_letters(a, b):
    return set(a.lower()) == set(b.lower())
print(is_same_letters("listen", "silent"))  # True

# 5. Schema Validation Architecture
def validate_form(data, required):
    missing = required - data.keys()
    if missing:
        return f"Missing fields: {missing}"
    return "OK"

required = {"name", "email", "age"}
form = {"name": "Alice", "email": "a@b.com", "city": "Delhi"}
print(validate_form(form, required))   # Missing fields: {'age'}

```

```python
# 6. Graph Structures: Tracking Network Reachability via Constant Sets
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[], 5:[]}
def reachable(start, graph):
    visited = {start}
    queue   = [start]
    while queue:
        node = queue.pop(0)
        for nb in graph[node]:
            if nb not in visited:      # Fast constant evaluation barrier
                visited.add(nb)
                queue.append(nb)
    return visited
print(reachable(1, graph))   # {1, 2, 3, 4, 5}

# 7. Combinatorial Generation: Designing Power Sets via Immutable Composition
def power_set(s):
    ps = {frozenset()}
    for elem in s:
        ps = ps | {sub | {elem} for sub in ps}
    return ps
print(power_set({1,2,3}))
```

2.5.3.4 Collection Types — At a Glance

Python Built-in Collection Types Summary

	list	tuple	set	frozenset	dict
O	Yes	Yes	No	No	Yes
M	Yes	No	Yes	No	Yes
Dup	Yes	Yes	No	No	Keys:No
Ha	No	Yes*	No	Yes	No
O(1) I	No	No	Yes	Yes	Yes (key)

** tuple is hashable only if all elements are hashable*

2.6 Problems

Problem 53 ***[MCQ]*** *What is the output of the following code?*

```
s = "python"
s[0] = "P"
print(s)
```

(A) `Python`

(B) `python`

(C) `TypeError`, *because strings are immutable*

(D) `IndexError`

Problem 54 ***[MCQ]*** *What is the output?*

```
s = "abcdef"
print(s[1:4])
```

(A) `abc`

(B) `bcd`

(C) `bcde`

(D) `abcd`

Problem 55 ***[MCQ]*** *What is the output?*

```
s = "hello"
print(s[::-1])
```

(A) `hello`

(B) `olleh`

(C) `hllo`

(D) `ello`

Problem 56 ***[MCQ]*** *What is the output?*

```
s = "abcdefgh"
print(s[::2])
```

(A) `aceg`

(B) `bdfh`

(C) `abcd`

(D) `efgh`

Problem 57 ***[MCQ]*** *What is the output?*

```
s = "gate2025"
print(s[-4:])
```

(A) `gate`

(B) `2025`

(C) `025`

(D) `e202`

Problem 58 ***[NAT]*** *What is the length of the string returned by* `"abcde"[1:4]`*?*

Problem 59 ***[MCQ]*** *What does `r"\n"` represent in Python?*

(A) A newline character

(B) The two-character sequence backslash followed by `n`

(C) An empty string

(D) `SyntaxError`

Problem 60 ***[MCQ]*** *What is the output?*

```
x = 3.14159
print(f"{x:.2f}")
```

(A) `3.14159`

(B) `3.14`

(C) `3.1`

(D) `3.142`

Problem 61 ***[MCQ]*** *What is the output?*

```
s = "hello world"
print(s.find("world"))
```

(A) `0`

(B) `5`

(C) `6`

(D) `-1`

Problem 62 ***[MCQ]*** *What is the difference between `str.find()` and `str.index()`?*

(A) `find()` is case-insensitive; `index()` is case-sensitive.

(B) `find()` returns -1 if not found; `index()` raises `ValueError`.

(C) `find()` returns a bool; `index()` returns an int.

(D) There is no difference.

Problem 63 ***[MCQ]*** *What is the output?*

```
s = "banana"
print(s.count("an"))
```

(A) `1`

(B) `2`

(C) `3`

(D) `0`

Problem 64 ***[MCQ]*** *What is the output?*

```
s = "  hello  "
print(repr(s.strip()))
```

(A) `' hello '`

(B) `'hello'`

(C) `'hello '`

(D) `' hello'`

Problem 65 ***[MCQ]*** *What is the output?*

```
words = ["py", "thon", "is", "fun"]
print("-".join(words))
```

(A) `py-thon-is-fun`

(B) `['py', 'thon', 'is', 'fun']`

(C) `python is fun`

(D) `TypeError`

Problem 66 ***[MCQ]*** *What is the output?*

```
s = "one,two,,three"
print(s.split(","))
```

(A) `['one', 'two', 'three']`

(B) `['one', 'two', '', 'three']`

(C) `['one,two,,three']`

(D) `('one', 'two', '', 'three')`

Problem 67 ***[MCQ]*** *What is the output?*

```
s = "hello@world.com"
print(s.partition("@"))
```

(A) `['hello', '@', 'world.com']`

(B) `('hello', '@', 'world.com')`

(C) `('hello', 'world.com')`

(D) `'hello world.com'`

Problem 68 ***[MCQ]*** *What is the output?*

```
s = "abcabc"
print(s.replace("b", "X", 1))
```

(A) `aXcaXc`

(B) `aXcabc`

(C) `abcaXc`

(D) `aXc`

Problem 69 ***[MSQ]*** *Which of the following string methods return a* ***new string*** *(do not modify in-place)?*

(A) `upper()`

(B) `replace()`

(C) `strip()`

(D) `split()`

Problem 70 ***[NAT]*** *What is the output of* `len("hello\nworld")`*? (Count actual characters, where* `\n` *is one character.)*

Problem 71 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3]
L.append([4, 5])
print(len(L))
```

(A) 4

(B) 5

(C) 3

(D) 6

Problem 72 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3]
L.extend([4, 5])
print(L)
```

(A) [1, 2, 3, [4, 5]]

(B) [1, 2, 3, 4, 5]

(C) [[1, 2, 3], [4, 5]]

(D) [4, 5]

Problem 73 ***[MCQ]*** *What is the output?*

```
L = [10, 20, 30, 40, 50]
print(L[1:4:2])
```

(A) [20, 30, 40]

(B) [20, 40]

(C) [10, 30, 50]

(D) [30]

Problem 74 ***[MCQ]*** *What is the output?*

```
L = [3, 1, 4, 1, 5, 9, 2]
L.sort()
print(L)
```

(A) [9, 5, 4, 3, 2, 1, 1]

(B) [1, 1, 2, 3, 4, 5, 9]

(C) [3, 1, 4, 1, 5, 9, 2]

(D) None

Problem 75 ***[MCQ]*** *What is the output?*

```
L = [3, 1, 4, 1, 5, 9, 2]
print(L.sort())
```

(A) [9, 5, 4, 3, 2, 1, 1]

(B) [1, 1, 2, 3, 4, 5, 9]

(C) [3, 1, 4, 1, 5, 9, 2]

(D) None

Problem 76 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3, 4, 5]
x = L.pop(1)
print(x, L)
```

(A) 1 [2, 3, 4, 5]

(B) `2 [1, 3, 4, 5]`

(C) `5 [1, 2, 3, 4]`

(D) `2 [1, 2, 3, 4, 5]`

Problem 77 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3, 2, 4]
L.remove(2)
print(L)
```

(A) `[1, 3, 2, 4]`

(B) `[1, 3, 4]`

(C) `[1, 2, 3, 4]`

(D) `ValueError`

Problem 78 ***[MCQ]*** *What is the output?*

```
L = [0] * 3
L[0].append(1)
print(L)
```

(A) `[1, 0, 0]`

(B) `[[1], [1], [1]]`

(C) `AttributeError`, because `int` has no `append`

(D) `[[1], 0, 0]`

Problem 79 ***[MCQ]*** *What is the output?*

```
L = [[0] * 2] * 3
L[0][0] = 9
print(L)
```

(A) `[[9, 0], [0, 0], [0, 0]]`

(B) `[[9, 0], [9, 0], [9, 0]]`

(C) `[[0, 0], [0, 0], [0, 0]]`

(D) `[[9, 9], [9, 9], [9, 9]]`

Problem 80 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3]
M = L[:]
M.append(4)
print(L, M)
```

(A) `[1, 2, 3, 4] [1, 2, 3, 4]`

(B) `[1, 2, 3] [1, 2, 3, 4]`

(C) `[1, 2, 3] [1, 2, 3]`

(D) `[1, 2, 3, 4] [1, 2, 3]`

Problem 81 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3,[10,20]]

M = L[:]

M[3].append(30)

print(L, M)
```

(A) `[1, 2, 3, [10, 20, 30]] [1, 2, 3, [10, 20, 30]]`

(B) `[1, 2, 3,[10,20]] [1, 2, 3, [10, 20, 30]]`

(C) `[1, 2, 3,[10,20]] [1, 2, 3, [10, 20]]`

(D) `[1, 2, 3] [1, 2, 3, 30]`

Problem 82 ***[MCQ]*** *What is the output?*

```
L = [1, [2, 3], 4]
import copy
M = copy.deepcopy(L)
M[1].append(99)
print(L[1])
```

(A) `[2, 3, 99]`

(B) `[2, 3]`

(C) `[99]`

(D) `None`

Problem 83 ***[MSQ]*** *Which of the following create a* ***shallow copy*** *of list* `L`*?*

(A) `L[:]`

(B) `L.copy()`

(C) `M = L`

(D) `copy.deepcopy(L)`

Problem 84 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3, 4, 5]
L[1:3] = [20, 30, 40]
print(L)
```

(A) `[1, 20, 30, 40, 4, 5]`

(B) `[1, 20, 30, 40, 5]`

(C) `[1, 20, 30, 3, 4, 5]`

(D) `[20, 30, 40]`

Problem 85 ***[MCQ]*** *What is the output?*

```
L = [5, 3, 8, 1]
print(sorted(L), L)
```

(A) `[1, 3, 5, 8] [1, 3, 5, 8]`

(B) `[1, 3, 5, 8] [5, 3, 8, 1]`

(C) `None [1, 3, 5, 8]`

(D) `[5, 3, 8, 1] [5, 3, 8, 1]`

Problem 86 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3]
L += [4, 5]
print(L)
```

(A) `[1, 2, 3, [4, 5]]`

(B) `[1, 2, 3, 4, 5]`

(C) `[[1, 2, 3], [4, 5]]`

(D) `[4, 5]`

Problem 87 ***[NAT]*** *What is the output of* `[1, 2, 3].count(2)`*?*

Problem 88 ***[MCQ]*** *What is the type of* `t = (5,)`*?*

(A) `int`

(B) `tuple`

(C) `list`

(D) `SyntaxError`

Problem 89 ***[MCQ]*** *What is the type of* `t = (5)`*?*

(A) `tuple`

(B) `int`

(C) `list`

(D) `SyntaxError`

Problem 90 ***[MCQ]*** *What is the output?*

```
t = (1, 2, 3, 4, 5)
first, *middle, last = t
print(middle)
```

(A) `(2, 3, 4)`

(B) `[2, 3, 4]`

(C) `[1, 2, 3, 4]`

(D) `(1, 2, 3, 4)`

Problem 91 ***[MCQ]*** *What is the output?*

```
t = (1, [2, 3], 4)
t[1].append(99)
print(t)
```

(A) `TypeError`, *because tuples are immutable*

(B) `(1, [2, 3, 99], 4)`

(C) `(1, [2, 3], 4)`

(D) `(1, [99], 4)`

Problem 92 ***[MSQ]*** *Which of the following are valid reasons to* ***prefer a tuple over a list****?*

(A) *Tuples can be used as dictionary keys (when all elements are hashable).*

(B) *Tuples are slightly more memory-efficient.*

(C) *Tuples support more methods than lists.*

(D) *Tuples can be elements of a set.*

Problem 93 ***[MCQ]*** *What is the output?*

```
(a, b), c = (1, 2), 3
print(a, b, c)
```

(A) `1 2 3`

(B) `(1, 2) 3`

(C) `1 (2, 3)`

(D) `ValueError`

Problem 94 ***[MCQ]*** *What is the output?*

```
t = (3, 1, 4, 1, 5, 1)
print(t.count(1), t.index(4))
```

(A) `3 3`

(B) `3 2`

(C) `2 2`

(D) `1 2`

Problem 95 ***[MCQ]*** *Which of the following will raise a* `TypeError`*?*

(A) Using `([1,2], 3)` *as a dictionary key*

(B) Using `(1, 2, 3)` *as a dictionary key*

(C) Using `("a", "b")` *as a set element*

(D) Using `(1, (2, 3))` *as a dictionary key*

Problem 96 ***[MCQ]*** *What is the output?*

```
d = {"a": 1, "b": 2}
print(d["c"])
```

(A) `None`

(B) `0`

(C) `KeyError`

(D) `AttributeError`

Problem 97 ***[MCQ]*** *What is the output?*

```
d = {"a": 1, "b": 2}
print(d.get("c", 99))
```

(A) `KeyError`

(B) `None`

(C) `99`

(D) `0`

Problem 98 ***[MCQ]*** *What is the output?*

```
d = {"x": 10}
d.setdefault("x", 99)
d.setdefault("y", 99)
print(d)
```

(A) `{'x': 99, 'y': 99}`

(B) `{'x': 10, 'y': 99}`

(C) `{'x': 10}`

(D) `{'y': 99}`

Problem 99 ***[MCQ]*** *What is the output?*

```
d = {"a": 1, "b": 2, "c": 3}
d.update({"b": 20, "d": 4})
print(d)
```

(A) {'a': 1, 'b': 2, 'c': 3}

(B) {'a': 1, 'b': 20, 'c': 3, 'd': 4}

(C) {'b': 20, 'd': 4}

(D) KeyError

Problem 100 ***[MCQ]*** *What is the output?*

```
d = {"a": 1, "b": 2}
val = d.pop("b")
print(val, d)
```

(A) 2 {'a': 1}

(B) None {'a': 1}

(C) 1 {'b': 2}

(D) KeyError

Problem 101 ***[MCQ]*** *What is the output?*

```
d = {"p": 1, "q": 2, "r": 3}
print(list(d.values()))
```

(A) ['p', 'q', 'r']

(B) [1, 2, 3]

(C) [('p', 1), ('q', 2), ('r', 3)]

(D) dict_values([1, 2, 3])

Problem 102 ***[MSQ]*** *Consider the following Python code snippet:*

```
d = \{"a": 1, "b": 2\}

k\_view = d.keys()

v\_view = d.values()

i\_view = d.items()

d["c"] = 3
```

Which of the following statements are ***correct*** *regarding the state of the view objects after the dictionary is mutated?*

(A) `list(k_view)` evaluates to ['a', 'b', 'c']

(B) `list(v_view)` evaluates to [1, 2, 3]

(C) `list(i_view)` evaluates to [('a', 1), ('b', 2), ('c', 3)]

(D) Modifying the underlying dictionary does not affect previously created view objects, so `list(k_view)` remains ['a', 'b']

Problem 103 ***[MSQ]*** *Which of the following are* ***valid dictionary keys*** *in Python?*

(A) "name"

(B) (1, 2)

(C) [1, 2]

(D) {1, 2}

Problem 104 ***[MCQ]*** *What is the output?*

```
d = dict.fromkeys(["a", "b", "c"], 0)
print(d)
```

(A) `{'a': None, 'b': None, 'c': None}`

(B) `{'a': 0, 'b': 0, 'c': 0}`

(C) `[('a', 0), ('b', 0), ('c', 0)]`

(D) `KeyError`

Problem 105 ***[MCQ]*** *What is the output?*

```
d1 = {"a": 1}
d2 = {"b": 2}
d3 = d1 | d2
print(d3)
```

(A) `{'a': 1}`

(B) `{'b': 2}`

(C) `{'a': 1, 'b': 2}`

(D) `TypeError`

Problem 106 ***[MCQ]*** *What is the output?*

```
d = {"a": {"x": 10}, "b": 2}
d["a"]["y"] = 20
print(d)
```

(A) `{'a': {'x': 10}, 'b': 2}`

(B) `{'a': {'x': 10, 'y': 20}, 'b': 2}`

(C) `KeyError`

(D) `{'a': {'y': 20}, 'b': 2}`

Problem 107 ***[NAT]*** *What is the output of the following code?*

```
d = {"a": 1, "b": 2, "c": 3}
print(len(d.keys()))
```

Problem 108 ***[MCQ]*** *What is the output?*

```
s = {}
print(type(s))
```

(A) `<class 'set'>`

(B) `<class 'dict'>`

(C) `<class 'frozenset'>`

(D) `<class 'NoneType'>`

Problem 109 ***[MCQ]*** *What is the output?*

```
s = {1, 2, 2, 3, 3, 3}
print(len(s))
```

(A) `6`

(B) `3`

(C) 2

(D) 1

Problem 110 ***[MCQ]*** *What is the output?*

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A & B)
```

(A) {1, 2, 3, 4, 5, 6}

(B) {3, 4}

(C) {1, 2}

(D) {1, 2, 5, 6}

Problem 111 ***[MCQ]*** *What is the output?*

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A ^ B)
```

(A) {3, 4}

(B) {1, 2, 5, 6}

(C) {1, 2, 3, 4, 5, 6}

(D) {1, 2}

Problem 112 ***[MCQ]*** *What is the output?*

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A - B)
```

(A) {1, 2}

(B) {5, 6}

(C) {3, 4}

(D) {1, 2, 5, 6}

Problem 113 ***[MCQ]*** *What is the difference between* `set.remove(x)` *and* `set.discard(x)`*?*

(A) `remove()` deletes all occurrences; `discard()` deletes only one.

(B) `remove()` raises `KeyError` if x is absent; `discard()` does not.

(C) `discard()` raises `KeyError` if x is absent; `remove()` does not.

(D) There is no difference.

Problem 114 ***[MCQ]*** *What is the output?*

```
A = {1, 2, 3}
B = {1, 2}
print(B <= A, B < A)
```

(A) False False

(B) True False

(C) True True

(D) False True

Problem 115 ***[MCQ]*** *What is the output?*

```
s = {3, 1, 4, 1, 5}
s.add(4)
s.add(9)
print(len(s))
```

(A) `6`

(B) `5`

(C) `4`

(D) `7`

Problem 116 ***[MSQ]*** *Which of the following can be added to a Python* `set`*?*

(A) `"hello"`

(B) `(1, 2)`

(C) `[1, 2]`

(D) `frozenset({1, 2})`

Problem 117 ***[MCQ]*** *What is the output?*

```
A = {1, 2, 3}
B = {4, 5, 6}
print(A.isdisjoint(B))
```

(A) `False`

(B) `True`

(C) `set()`

(D) `TypeError`

Problem 118 ***[MCQ]*** *What is the output?*

```
fs = frozenset([1, 2, 3])
fs.add(4)
print(fs)
```

(A) `frozenset({1, 2, 3, 4})`

(B) `{1, 2, 3, 4}`

(C) `AttributeError`

(D) `TypeError`

Problem 119 ***[MCQ]*** *What is the output?*

```
d = {"a": [1, 2], "b": [3, 4]}
e = d.copy()
e["a"].append(99)
print(d["a"])
```

(A) `[1, 2]`

(B) `[1, 2, 99]`

(C) `[99]`

(D) `KeyError`

Problem 120 ***[MSQ]*** *Which of the following statements about Python's built-in data structures are* ***correct****?*

(A) Lists are ordered and mutable.

(B) Dictionaries are ordered by insertion order (Python 3.7+).

(C) Sets maintain insertion order.

(D) Tuples are immutable but can contain mutable objects.

(E) Frozensets are hashable and can be used as dictionary keys.

Problem 121 ***[MCQ]*** *What is the output?*

```
L = [1, 2, 3, 4, 5]
del L[::2]
print(L)
```

(A) `[2, 4]`

(B) `[1, 3, 5]`

(C) `[2, 3, 4]`

(D) `[1, 2, 4]`

Problem 122 ***[MSQ]*** *Consider the following Python code execution snippets:*

```
A = {1, 2, 3}
B = [3, 4, 5]
```

Which of the following execution lifecycles will run successfully without raising a `TypeError`*?*

(A) `A | B`

(B) `A.union(B)`

(C) `A |= B`

(D) `A.update(B)`

Problem 123 ***[MSQ]*** *Let A and B be two distinct, non-empty Python* `set` *instances. Which of the following operations will allocate a brand-new* `set` *object in memory rather than modifying an existing object in-place?*

(A) `A.difference(B)`

(B) `A -= B`

(C) `A.symmetric_difference_update(B)`

(D) `A ^ B`

Problem 124 ***[MSQ]*** *Which of the following structural relations or mutations between two Python sets can* ***only*** *be expressed using Python operators, due to the deliberate absence of a corresponding named method in the language specification?*

(A) Checking if set A is a proper subset of set B.

(B) Performing an in-place intersection update on set A using set B.

(C) Checking if set A is a proper superset of set B.

(D) Computing the symmetric difference between set A and set B.

Problem 125 ***[MSQ]*** *We wish to determine if two sets, A and B, share no elements in common (i.e., they are disjoint). Which of the following syntax structures can be natively executed in Python to perform this check?*

(A) `A.isdisjoint(B)`

(B) `A == B`

(C) `not (A & B)`

(D) `A.intersection(B) == set()`

Problem 126 ***[MSQ]*** *Which of the following options correctly describe constraints or structural behaviors of in-place operations in Python?*

(A) *In-place methods like `A.difference_update(B)` return `None` upon execution.*

(B) *There are no dedicated in-place operators or in-place methods for evaluating subset or superset relations.*

(C) *The statement `A &= B` is functionally valid even if `B` is a standard Python dictionary.*

(D) *In-place operators modify the original object's hash values, causing them to be re-indexed if stored inside another collection.*

Problem 127 ***[MCQ]*** *What is the output of the following Python code snippet?*

```
s = \{1, 2, 3\}
fs = frozenset([3, 4, 5])

res1 = type(s & fs)
res2 = type(fs & s)

print(res1 == res2)
```

(A) `True`

(B) `False`

(C) `TypeError`

(D) `None`

Problem 128 ***[MSQ]*** *Consider the following Python expressions operating on a standard set `s` and a frozenset `fs`:*

```
s = \{1, 2\}
fs = frozenset([2, 3])
```

*Which of the following operations will produce an output object that is fully **hashable** (and can therefore be safely used as a dictionary key)?*

(A) `s | fs`

(B) `s.union(fs)`

(C) `fs | s`

(D) `fs.intersection(s)`

2.7 YouTube Links and QR Codes

Lecture	Details	YouTube Link	QR Code
06	Strings in Python	https://youtu.be/6q1UlqlEyDo	
07	Lists in Python	https://youtu.be/d94o-4Dhti8	
08	Tuples in Python	https://youtu.be/EwwlWZ1ZSgk	
09	Dictionaries in Python	https://youtu.be/G1JSU0mrecs	
10	Sets & Frozensets in Python	https://youtu.be/aymyBSXgel8	

11	Problem Solving on Strings, Lists, Tuples, Dictionaries & Sets	`https://youtu.be/TzahIwnXL2U`	

Chapter 3

Control Flow

3.1 Conditional Statements

3.1.1 if / elif / else

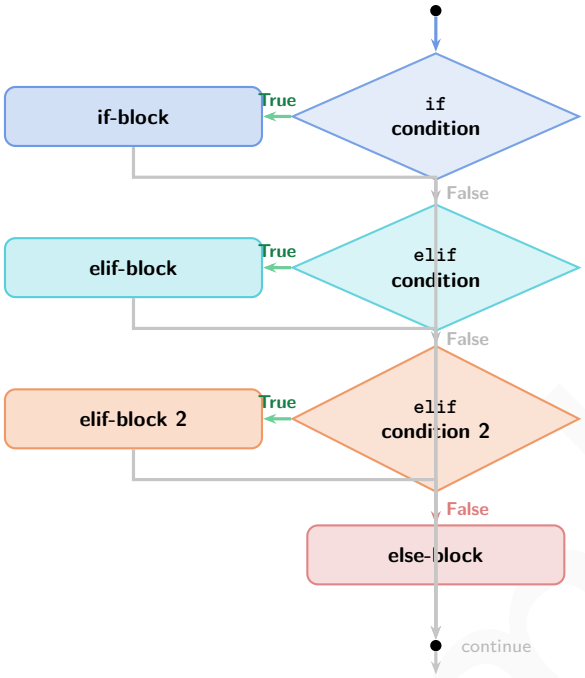
3.1.1.1 Syntax, Chaining and Nesting

if / elif / else — Formal Syntax

Python conditional execution uses **if**, **elif** (else-if), and **else**. Only the *first true* branch executes; the rest are skipped.

Clause	Rule
`if`	Required; exactly one per block
`elif`	Optional; zero or more; checked in order
`else`	Optional; exactly one; catches all remaining cases

Evaluation: conditions are tested *top-to-bottom*; as soon as one is `True` its block executes and *all remaining elif/else are skipped*.



if / elif / else Properties

- **Short-circuit evaluation**: once a true branch is found, all subsequent `elif`/`else` blocks are *completely skipped* (conditions not even evaluated)
- **No switch/case in Python < 3.10**: use `if`/`elif` chains or a dict dispatch table
- **Python 3.10+**: `match`/`case` structural pattern matching (see below)
- **Any expression** as condition — Python evaluates its truthiness (`bool(expr)`)
- **One-liner**: `if x > 0: return x` allowed but use sparingly

Example 87: Basic if / elif / else

```
score = 83

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(grade)  # B

# Short-circuit: only first True branch runs
x = 15
if x > 10:
    print("Greater than 10")      # prints this
elif x > 5:                       # NOT evaluated (first was True)
    print("Greater than 5")
else:
    print("5 or less")
```

Example 88: Python 3.10+ match / case (Structural Pattern Matching)

```python
# match/case: Python 3.10+
point = (5, 0)

match point:
    case (0, 0):
        result = "Origin"
    case (x, 0):
        result = f"On x-axis at {x}"
    case (0, y):
        result = f"On y-axis at {y}"
    case (x, y):
        result = f"Point at ({x}, {y})"

print(result)  # On x-axis at 5

# match with guards (if condition)
value = "hi"

match value:
    case int(n) if n < 0:
        classification = "negative int"
    case int(n) if n == 0:
        classification = "zero"
    case int(n):
        classification = "positive int"
    case str(s):
        classification = f"string of length {len(s)}"
    case _:               # wildcard: matches anything
        classification = "other"

print(classification)  # string of length 2
```

3.1.2 Ternary (Conditional) Expression

3.1.2.1 Inline if-else

Ternary Expression — Syntax and Evaluation

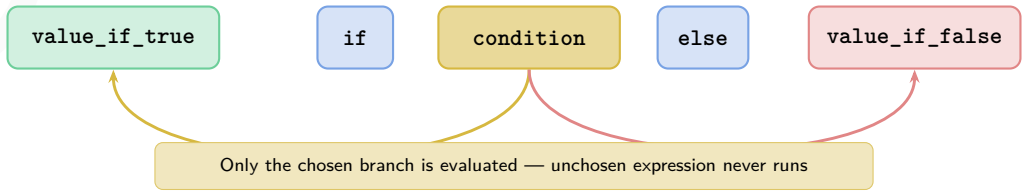
Python's **conditional expression** (ternary operator) evaluates to one of two values based on a condition, in a single expression:

`value_if_true if condition else value_if_false`

Evaluation order:

1. Condition is evaluated first
2. If `True`: `value_if_true` is evaluated and returned
3. If `False`: `value_if_false` is evaluated and returned
4. The *unchosen* branch is **not evaluated** (lazy/short-circuit)

Contrast with C/Java ternary: `cond ? a : b`
Python uses `a if cond else b` (reads more like English).



Ternary Properties and Patterns

- **Returns a value**: can be used anywhere an expression is needed (assignment, function argument, list comprehension, f-string)
- **Lazy**: only the selected branch is evaluated (`1/0 if False else 99` → 99, no `ZeroDivisionError`)
- **Nesting allowed but discouraged**: `a if p else b if q else c` is valid but hard to read
- **Common pattern**: safe default — `x if x is not None else 0`
- **vs `or` pattern**: `x or default` has a subtle difference — `x or default` returns `default` when `x` is *any falsy value* (including 0, `""`), not just `None`

Example 89: Ternary Expression — All Use Cases

```python
# Basic usage
x = 10
result = "positive" if x > 0 else "non-positive"
print(result)   # positive

# In assignment
score = 85
grade = "Pass" if score >= 60 else "Fail"
print(grade)    # Pass

# Inside a print call argument
print(f"Status: {'Active' if True else 'Inactive'}")   # Status: Active

# In f-string
n = 7
print(f"{n} is {'odd' if n % 2 else 'even'}")     # 7 is odd

# In list comprehension
data = [-3, -1, 0, 2, 5, -4, 8]
abs_vals = [x if x >= 0 else -x for x in data]
print(abs_vals)   # [3, 1, 0, 2, 5, 4, 8]

# Lazy evaluation: unchosen branch not evaluated
safe = 1/0 if False else 99   # no ZeroDivisionError!
print(safe)   # 99

# Nested ternary (valid, but avoid for readability)
x = 0
sign = "positive" if x > 0 else ("negative" if x < 0 else "zero")
print(sign)   # zero

# PITFALL: or vs ternary for None check
val = 0
print(val or "default")              # "default"  (0 is falsy!)
print(val if val is not None else "default")  # 0  (correct None-check)
```

3.1.3 Truthy / Falsy Values

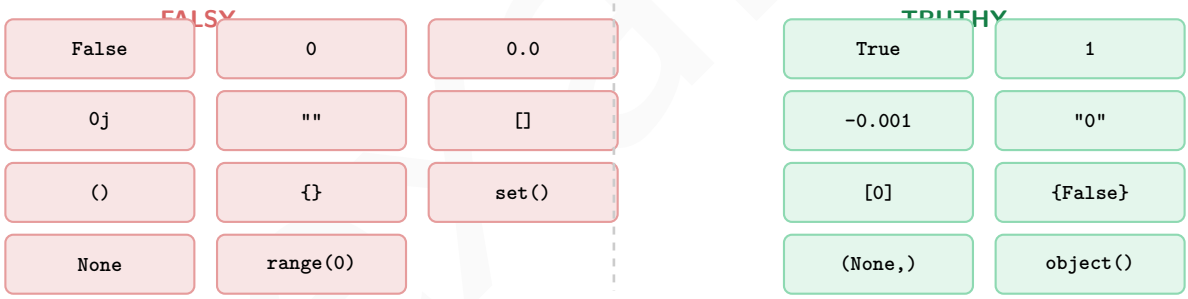
3.1.3.1 Python's Boolean Context

Truthiness in Python — Complete Rules

Every Python object has an implicit boolean value used in `if`, `while`, `and`, `or`, `not` contexts.
Falsy values — evaluate to `False` in boolean context:

Value	Type	Note
`False`	`bool`	The boolean False
`0`	`int`	Zero integer
`0.0`	`float`	Zero float
`0j`	`complex`	Zero complex
`""`	`str`	Empty string
`[]`	`list`	Empty list
`()`	`tuple`	Empty tuple
`{}`	`dict`	Empty dict
`set()`	`set`	Empty set
`None`	`NoneType`	The null singleton
`range(0)`	`range`	Empty range

Truthy: everything else — non-zero numbers, non-empty containers, most objects (including `0.0001`, `"0"`, `[0]`, `{False}`).
Custom objects: define `__bool__` (returns `True`/`False`) or `__len__` (falsy if `len == 0`).



Example 90: Truthiness — Pitfalls and Correct Idioms

```
# Checking empty containers the Pythonic way
lst = []
if not lst:            # Pythonic: rely on truthiness
    print("empty")

# Anti-pattern (verbose)
if len(lst) == 0:      # works but not idiomatic
    print("empty")

# PITFALL: "0" is truthy (non-empty string)
s = "0"
if s:
    print("truthy!")  # prints! "0" is NOT the int 0

# PITFALL: [0] is truthy (non-empty list)
L = [0]
if L:
    print("truthy!")  # prints! list is non-empty

# PITFALL: {False} is truthy (non-empty set)
st = {False}
```

```python
if st:
    print("truthy!")  # prints! set is non-empty

# None vs falsy: use 'is None' to distinguish
a, b = 10, 0
result = a / b if b else None    # None on zero

if result is None:           # correct None check
    print("Division by zero")
if not result:               # WRONG: also catches result=0.0
    print("Would also catch 0.0!")
```

3.2 Loops

3.2.1 for Loop

3.2.1.1 Iterating Iterables, range, enumerate, zip

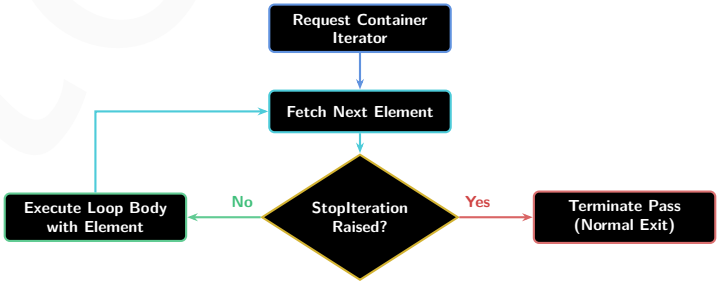
for Loop — Repeating Things Easily

A **for loop** is like a train track that visits every single station on a line. It tells Python to take a collection of items (like a list of toys or letters in a word) and look at each item, one by one, from the first to the last.

`for` ⟨*item*⟩ `in` ⟨*collection*⟩`:` ⟨*what to do with it*⟩

Things you can loop through include: `str` (letters in text), `list` (a list of things), `tuple` (fixed list), `set` (unique items), `dict` (dictionary keys), and helper functions like `range`, `enumerate`, and `zip`.
range() parameters (How to count):

Type	What it counts	Example
`range(stop)`	Starts at 0, stops just before `stop`	`range(5)` → 0, 1, 2, 3, 4
`range(start, stop)`	Starts at `start`, stops before `stop`	`range(2,6)` → 2, 3, 4, 5
`range(start, stop, step)`	Skips numbers by the `step` size	`range(0,10,2)` → 0, 2, 4, 6, 8
`range(10, 0, -1)`	Counts down backwards	10, 9, 8,. . . , 1



Cool Loop Assistants: enumerate() and zip()

- `enumerate(items, start=0)`: Tells you the item *and* its tracking number (index). Like counting: "Item number 0 is an apple, item number 1 is a banana."
- `zip(list1, list2, ...)`: Pairs things up like matching socks. If you have a list of names and a list of ages, it matches the 1st name with the 1st age. It stops as soon as the *shortest* list runs out!
- `itertools.zip_longest()`: Keeps pairing even if one list is longer, filling the empty spots with a placeholder (like `None` or 0).

- `zip(*matrix)`: Swaps rows and columns in a grid (like turning a tall photo into a wide photo).
- **range** is smart and lazy: It doesn't actually write a massive list of numbers in the computer's memory. Even `range(10**9)` takes up almost no space!

What is `end=""`?

By default, every time Python uses `print()`, it automatically jumps to a **new line** (like pressing the Enter key on a keyboard).
When you use `end=""`, you tell Python: *"Don't jump to a new line! Keep the printing cursor right where it is so the next print stays on the same row."*

Example 91: Comparing Normal Print vs. `end=""`

```python
# 1. Without end="" (Default - prints downward)
print("Apple")
print("Banana")
# Output:
# Apple
# Banana

# 2. With end="" (Glues things together side-by-side)
print("Apple", end="")
print("Banana", end="")
# Output: AppleBanana

# 3. Adding a space inside end=" " (Keeps things neat in a row)
for i in range(3):
    print("Hi", end=" ")
# Output: Hi Hi Hi

# 4. Adding a symbol inside end="-"
for i in range(3):
    print("Go", end="-")
# Output: Go-Go-Go-
```

Example 92: for Loop — Simple Examples

```python
# 1. Counting up to 5 (stops before 5)
for i in range(5):
    print(i, end=" ")   # 0 1 2 3 4

# 2. Skipping numbers (count by twos)
for i in range(2, 8, 2):
    print(i, end=" ")   # 2 4 6

# 3. Rocket countdown!
for i in range(5, 0, -1):
    print(i, end=" ")   # 5 4 3 2 1

# 4. Spell a word letter by letter
for letter in "CAT":
    print(letter, end=" ")   # C A T

# 5. Looking at keys in a dictionary
toy_box = {"car": "red", "ball": "blue"}
for toy in toy_box:
    print(toy, end=" ")   # car ball

# 6. Using enumerate to get numbers
fruits = ["apple", "banana"]
for index, fruit in enumerate(fruits):
    print(f"Item {index} is {fruit}")
# Item 0 is apple
```

```python
# Item 1 is banana

# 7. Using zip to pair friends with their favorite colors
names  = ["Alice", "Bob"]
colors = ["Red", "Blue"]
for name, color in zip(names, colors):
    print(f"{name} likes {color}")
# Alice likes Red
# Bob likes Blue

# 8. zip stops when the shorter list is empty
numbers = [1, 2, 3, 4, 5]
letters = ["A", "B"]
print(list(zip(numbers, letters)))    # [(1, 'A'), (2, 'B')]

# 9. zip_longest keeps going to the end
from itertools import zip_longest
print(list(zip_longest(numbers, letters, fillvalue="Empty")))
# [(1, 'A'), (2, 'B'), (3, 'Empty'), (4, 'Empty'), (5, 'Empty')]
```

Example 93: Common Loop Tricks

```python
# 1. Adding up numbers (Piggy bank)
coins = [5, 2, 10, 1]
total = 0
for coin in coins:
    total += coin
print(total)   # 18

# 2. Loop inside a loop (Making a multiplication grid)
for row in range(1, 3):
    for col in range(1, 3):
        print(f"{row}x{col}={row*col}", end="  ")
    print() # Moves to the next line
# 1x1=1  1x2=2
# 2x1=2  2x2=4

# 3. Reading dictionary keys and values together
pet_sounds = {"dog": "woof", "cat": "meow"}
for pet, sound in pet_sounds.items():
    print(f"The {pet} goes {sound}")

# 4. Making a new list using a loop
numbers = [1, 2, 3]
doubled = []
for x in numbers:
    doubled.append(x * 2)
print(doubled)   # [2, 4, 6]
```

3.2.2 while Loop

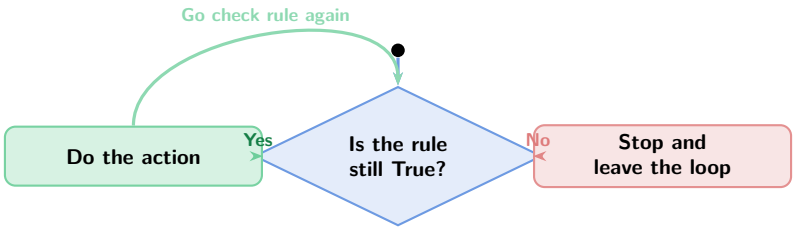
3.2.2.1 Condition-Controlled Repetition

while Loop — Repeat Until We Tell You to Stop

A **while loop** is like jumping on a trampoline. You keep jumping **while** you still have energy. You don't know how many times you will jump before you start, you just check if you are tired before every single jump.

`while` ⟨*rule is true*⟩: ⟨*keep doing this action*⟩

How it works: Python checks the rule *before* it does anything. If the rule is already `False` at the very start, the loop will never run even once!



Common while Loop Tricks and Traps

- **Counting Down**: `cookies = 5; while cookies > 0:  eat()`
- **Infinite Loop with a Break**: Using `while True:` creates a loop that runs forever, unless a special `break` command stops it inside.
- **Input Check**: Asking a user to type something until they type the correct answer (like entering a valid password).
- **The Danger Trap (Infinite Loop)**: If you forget to change something inside the loop, the rule stays `True` forever and the computer gets stuck in an infinite loop!

Example 94: while Loop Examples

```python
# 1. Simple countdown
candies = 3
while candies > 0:
    print(f"Eating a candy! {candies} left.")
    candies -= 1  # We eat one, so count goes down
# Eating a candy! 3 left.
# Eating a candy! 2 left.
# Eating a candy! 1 left.

# 2. Keep guessing until it's right
secret_number = 7
user_guess = 0  # Start with 0 so the loop begins

# Simulating guesses: 3, then 5, then 7
guesses = iter([3, 5, 7])

while user_guess != secret_number:
    user_guess = next(guesses)
    print(f"Guessed: {user_guess}")
    if user_guess == secret_number:
        print("You got it!")
    else:
        print("Wrong! Try again.")

# 3. THE INFINTIE LOOP PITFALL (What NOT to do!)
# item_count = 5
# while item_count > 0:
#     print("Stuck forever!")
#     # Uh oh! We forgot to do item_count -= 1, so it never stops!
```

3.2.3 Loop Control

3.2.3.1 break, continue, pass

break, continue, pass — Traffic Controls for Loops

Keyword	What it does	Where to use it
`break`	**Eject button:** Stops the loop completely right now.	`for` or `while`
`continue`	**Skip button:** Skips the rest of this turn and goes to the next turn.	`for` or `while`
`pass`	**Do nothing:** A blank placeholder that does absolutely nothing.	Anywhere

Important Note: `break` only gets you out of the *current closest* loop box. If you have a loop inside another loop, it won't escape the outer one!

Example 95: Break, Continue, and Pass Examples

```python
# 1. break: Stop searching when we find what we want
lucky_numbers = [1, 2, 9, 4, 5]
for num in lucky_numbers:
    if num == 9:
        print("Found the 9! Stopping the search.")
        break
    print(f"Checking: {num}")
# Checking: 1
# Checking: 2
# Found the 9! Stopping the search.

# 2. continue: Skip bad items (Skip even numbers)
for num in range(1, 6):
    if num % 2 == 0:
        continue # Skip this number!
    print(num, end=" ") # 1 3 5
print()

# 3. pass: Just a placeholder so Python doesn't get confused
for num in range(3):
    if num == 1:
        pass # I will write code here tomorrow!
    else:
        print(num, end=" ") # 0 2
```

3.2.4 else Clause on Loops

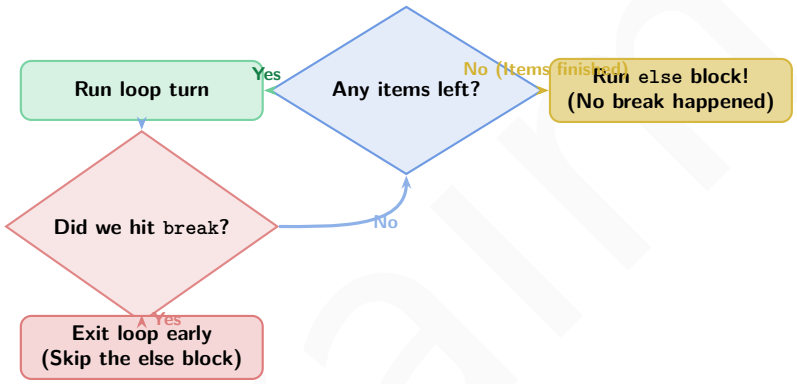
3.2.4.1 Loop–else: The "No Break" Reward

Loop else — What happens if we never hit break?

You can attach an `else` block to a loop. It acts like a reward block that runs **only if the loop finishes its entire run without getting stopped by a break**.

The else block runs if...	The loop finished naturally on its own.
The else block does NOT run if...	A `break` stopped the loop early.

Best way to remember: Think of it as a **"No Break Happened"** block. It is super useful for searching through things to confirm that something is missing.



Example 96: Loop else — Code Example

```python
# Look for a monster in the closet
closet = ["shirt", "pants", "socks"]

for item in closet:
    if item == "monster":
        print("Ahhh! Found a monster!")
        break
else:
    # This runs only if we checked EVERYTHING and found NO monster
    print("Yay! No monsters found in the closet.")

# Output: Yay! No monsters found in the closet.
```

3.2.5 Walrus Operator :=

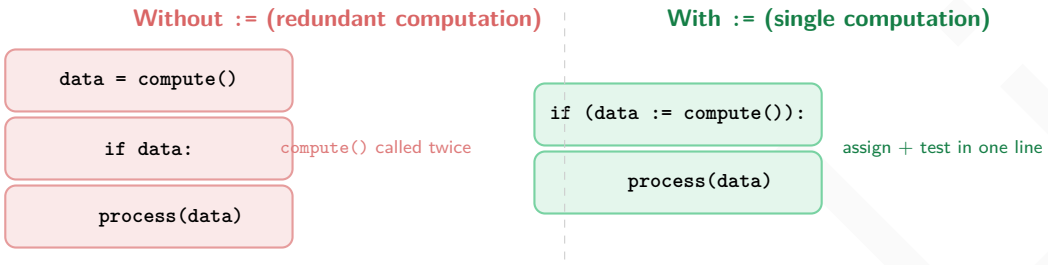
Walrus Operator := — Definition

The **walrus operator** (:=), introduced in Python 3.8 via PEP 572, is an **assignment expression**: it *assigns a value to a name* and *simultaneously returns that value* as an expression.

```
name := expression
```

This allows assignment to occur *inside* a condition, comprehension, or any other expression context where a plain = statement is not allowed.
Key rules:

- Must be parenthesised when used at the *top level of an expression statement* (`(x := 5)` not `x := 5` standalone)
- The assigned name is visible in the *enclosing scope* (not scoped to the comprehension variable)
- Cannot be used for attribute or subscript assignment (`obj.x := 5` is illegal)



Walrus Operator Use Cases

- **while loop**: `while chunk := f.read(4096):` — read until empty bytes
- **Regex match**: `if m := re.search(pat, text):` use `m.group()` in body
- **Comprehension filter and use**: compute expensive value once, filter and use in the same expression
- **Input loop**: `while (line := input("? ")) != "quit":`
- **Chain of transformations**: avoid intermediate variables while keeping result accessible

Example 97: Walrus Operator — All Patterns

```python
# 1. while loop: read file in chunks
# with open("data.bin", "rb") as f:
#     while chunk := f.read(4096):
#         process(chunk)

# Simulated version
import io
f = io.BytesIO(b"hello world this is a test")
chunks = []
while chunk := f.read(5):
    chunks.append(chunk)
print(chunks)
# [b'hello', b' worl', b'd thi', b's is ', b'a tes', b't']

# 2. Regex: assign match and test existence
import re
texts = ["Phone: 9876543210", "No phone here", "Call: 1234567890"]
for text in texts:
    if m := re.search(r'\d{10}', text):
        print(f"Found: {m.group()}")    # prints matched number
    else:
        print("No number in:", text)

# 3. Comprehension: avoid double computation
import math
data = [1, -2, 4, -9, 16, -25]
results = [root for x in data if (root := math.sqrt(x)) > 1.5]
# note: :=  inside comprehension; 'root' leaks to enclosing scope
print(results)    # [2.0, 4.0]

# 4. Input loop
prompts = iter(["hello", "world", "quit"])  # simulate input
words = []
while (word := next(prompts)) != "quit":
```

```
    words.append(word.upper())
print(words)   # ['HELLO', 'WORLD']

# 5. Avoid recomputing expensive call
def expensive(n): return n * n + n - 1

values = [1, 5, 3, 8, 2]
# Without walrus: expensive() called twice per iteration
filtered_bad = [expensive(x) for x in values if expensive(x) > 10]

# With walrus: computed once
filtered_good = [v for x in values if (v := expensive(x)) > 10]
print(filtered_good)   # [29, 61]

# 6. Nested while with walrus
total, count = 0, 0
data_stream = iter([4, 7, 2, -1, 5, -1])   # -1 is sentinel
while (val := next(data_stream)) != -1:
    total += val
    count += 1
print(f"Sum={total}, Count={count}, Mean={total/count:.2f}")
# Sum=13, Count=3, Mean=4.33
```

Example 98: Control Flow — Tricky Questions

```
# Q1: What does this print?
for i in range(3):
    for j in range(3):
        if j == 1:
            break
    print(i, j)
# break exits INNER loop; j=1 when break fires, but loop ends
# After inner loop, j retains last value (1 from break, NOT 2)
# Output: 0 1 / 1 1 / 2 1

# Q2: else on for
for i in range(5):
    if i == 10:   # never True
        break
else:
    print("no break")   # prints: no break

# Q3: continue vs pass
for i in range(5):
    if i == 2:
        pass          # does nothing; loop continues normally
    print(i, end=" ") # 0 1 2 3 4  (2 is still printed!)

for i in range(5):
    if i == 2:
        continue      # skips print for i=2
    print(i, end=" ") # 0 1 3 4    (2 is skipped!)

# Q4: walrus scope leak
data = [1, 2, 3, 4, 5]
evens = [y for x in data if (y := x*2) > 4]
print(y)   # 10  (last value assigned by := leaks out of comprehension!)

# Q5: while-else with continue
n = 10
while n > 0:
    n -= 1
    if n % 2 == 0:
        continue      # continue does NOT trigger else
else:
    print("while done")   # prints: while done (no break occurred)
```

3.3 Problems

Problem 129 ***[MCQ]*** *What is the output of the following code?*

```
x = 15
if x > 20:
    print("A")
elif x > 10:
    print("B")
elif x > 5:
    print("C")
else:
    print("D")
```

(A) `A`

(B) `B`

(C) `C`

(D) `B and C`

Problem 130 ***[MCQ]*** *What is the output?*

```
a, b = 5, 10
result = "yes" if a > b else "no"
print(result)
```

(A) `yes`

(B) `no`

(C) `True`

(D) `False`

Problem 131 ***[MCQ]*** *What is the output?*

```
x = 0
if x:
    print("truthy")
else:
    print("falsy")
```

(A) `truthy`

(B) `falsy`

(C) `0`

(D) `None`

Problem 132 ***[MCQ]*** *What is the output?*

```
x = 7
label = "high" if x > 10 else "mid" if x > 4 else "low"
print(label)
```

(A) `high`

(B) `mid`

(C) `low`

(D) `SyntaxError`

Problem 133 ***[MCQ]*** *What is the output?*

```
x = 3
if x == 1:
    print("one")
if x == 2:
    print("two")
if x == 3:
    print("three")
if x == 4:
    print("four")
```

(A) three

(B) No output

(C) three followed by four

(D) SyntaxError

Problem 134 ***[MCQ]*** *What is the output?*

```
val = None
print("exists" if val else "missing")
```

(A) exists

(B) missing

(C) None

(D) TypeError

Problem 135 ***[MSQ]*** *Which of the following are* ***falsy*** *in Python and would cause an* `if` *block to be skipped?*

(A) 0.0

(B) "0"

(C) []

(D) (0,)

(E) {}

Problem 136 ***[MCQ]*** *What is the output?*

```
x = 5
y = 10
z = 15
if x < y < z:
    print("ordered")
else:
    print("not ordered")
```

(A) not ordered

(B) ordered

(C) True

(D) SyntaxError

Problem 137 ***[MCQ]*** *What is the output?*

```
a = 4
b = "even" if a % 2 == 0 else "odd"
print(type(b), b)
```

(A) <class 'bool'> True

(B) <class 'str'> even

(C) `<class 'str'> odd`

(D) `<class 'int'> 0`

Problem 138 ***[NAT]*** *What is the output of the following code?*

```
total = 0
for i in range(1, 6):
    total += i
print(total)
```

Problem 139 ***[MCQ]*** *What is the output?*

```
for i in range(5, 0, -2):
    print(i, end=" ")
```

(A) `5 4 3 2 1`

(B) `5 3 1`

(C) `5 2`

(D) `4 2`

Problem 140 ***[MCQ]*** *What is the output?*

```
for i, v in enumerate(["a", "b", "c"], start=1):
    print(i, v, end=" ")
```

(A) `0 a 1 b 2 c`

(B) `1 a 2 b 3 c`

(C) `a 1 b 2 c 3`

(D) `(1, 'a') (2, 'b') (3, 'c')`

Problem 141 ***[MCQ]*** *What is the output?*

```
a = [1, 2, 3]
b = [4, 5]
for x, y in zip(a, b):
    print(x + y, end=" ")
```

(A) `5 7 3`

(B) `5 7`

(C) `5 7 ValueError`

(D) `1 2 3 4 5`

Problem 142 ***[NAT]*** *How many times is `print` called in the following code?*

```
for i in range(3):
    for j in range(4):
        print(i * j)
```

Problem 143 ***[MCQ]*** *What is the output?*

```
for i in range(10):
    if i % 2 == 0:
        continue
    if i > 6:
        break
    print(i, end=" ")
```

(A) `1 3 5`

(B) `1 3 5 7`

(C) `0 2 4 6`

(D) `1 3 5 7 9`

Problem 144 ***[MCQ]*** *What is the output?*

```
result = []
for i in range(1, 6):
    if i == 3:
        continue
    result.append(i)
print(result)
```

(A) `[1, 2, 3, 4, 5]`

(B) `[1, 2, 4, 5]`

(C) `[3]`

(D) `[1, 2]`

Problem 145 ***[MCQ]*** *What is the output?*

```
for i in range(3):
    pass
print(i)
```

(A) `0`

(B) `2`

(C) `3`

(D) `NameError`

Problem 146 ***[MCQ]*** *What does `zip` do when iterables have different lengths?*

(A) *It raises a `ValueError`.*

(B) *It pads the shorter iterable with `None`.*

(C) *It stops at the length of the* ***shortest*** *iterable.*

(D) *It stops at the length of the* ***longest*** *iterable.*

Problem 147 ***[MCQ]*** *What is the output?*

```
for i in range(0):
    print("inside")
print("done")
```

(A) `inside` *followed by* `done`

(B) `done`

(C) *No output*

(D) `ValueError`

Problem 148 ***[NAT]*** *What is the value of `total` after the following code executes?*

```
total = 0
for i in range(1, 11, 3):
    total += i
print(total)
```

Problem 149 ***[MCQ]*** *What is the output?*

```
pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
for num, ch in pairs:
    print(num, ch, end=" | ")
```

(A) (1, 'a') (2, 'b') (3, 'c')

(B) 1 a | 2 b | 3 c |

(C) 1 2 3 a b c

(D) ValueError

Problem 150 ***[MCQ]*** *What is the output?*

```
n = 1
while n < 32:
    n *= 2
print(n)
```

(A) 16

(B) 32

(C) 64

(D) 31

Problem 151 ***[MCQ]*** *What is the output?*

```
i = 10
while i > 0:
    i -= 3
print(i)
```

(A) 1

(B) 0

(C) -2

(D) 3

Problem 152 ***[MCQ]*** *What is the output?*

```
count = 0
i = 0
while i < 10:
    i += 1
    if i % 3 == 0:
        continue
    count += 1
print(count)
```

(A) 10

(B) 7

(C) 3

(D) 6

Problem 153 ***[MCQ]*** *What is the output?*

```
x = 5
while x > 0:
    print(x, end=" ")
    x -= 2
```

(A) 5 3 1

(B) `5 3 1 -1`

(C) `5 4 3 2 1`

(D) *Infinite loop*

Problem 154 ***[MCQ]*** *What is the output?*

```
i = 0
while True:
    i += 1
    if i == 5:
        break
print(i)
```

(A) `4`

(B) `5`

(C) `6`

(D) *Infinite loop — never prints*

Problem 155 ***[MCQ]*** *What is the output?*

```
for i in range(5):
    if i == 3:
        break
    print(i, end=" ")
```

(A) `0 1 2 3`

(B) `0 1 2`

(C) `1 2 3`

(D) `0 1 2 3 4`

Problem 156 ***[MCQ]*** *What is the output?*

```
for i in range(3):
    for j in range(3):
        if j == 1:
            break
        print(i, j)
```

(A) `0 0, 1 0, 2 0` *(one per line)*

(B) `0 0, 0 1, 1 0, 1 1, 2 0, 2 1`

(C) `0 0` *only*

(D) *No output*

Problem 157 ***[MCQ]*** *Which of the following is the correct description of* `pass` *in Python?*

(A) `pass` *exits the current loop iteration.*

(B) `pass` *terminates the enclosing loop.*

(C) `pass` *is a no-op; it does nothing and allows syntactically empty blocks.*

(D) `pass` *skips to the next iteration.*

Problem 158 ***[MCQ]*** *What is the output?*

```
for i in range(5):
    if i % 2 == 0:
        pass
    else:
        print(i, end=" ")
```

(A) `0 2 4`

(B) `1 3`

(C) `1 2 3`

(D) No output

Problem 159 ***[MSQ]*** *Which of the following statements about* `break` *are* ***correct****?*

(A) `break` *exits only the* ***innermost*** *enclosing loop.*

(B) `break` *can be used to exit multiple nested loops at once.*

(C) *When* `break` *is executed, the loop's* `else` *clause is* ***skipped****.*

(D) `break` *can appear outside of a loop.*

(E) *After* `break`*, execution continues at the statement following the loop.*

Problem 160 ***[MCQ]*** *What is the output?*

```
for i in range(5):
    if i == 3:
        break
else:
    print("completed")
print("done")
```

(A) `completed` followed by `done`

(B) `done`

(C) `completed`

(D) No output

Problem 161 ***[MCQ]*** *What is the output?*

```
for i in range(5):
    if i == 10:
        break
else:
    print("no break")
```

(A) No output

(B) `no break`

(C) `True`

(D) `SyntaxError`

Problem 162 ***[MCQ]*** *What is the output?*

```
target = 7
data = [1, 3, 5, 9, 11]
for x in data:
    if x == target:
        print("found")
        break
else:
    print("not found")
```

(A) `found`

(B) `not found`

(C) Both `found` and `not found`

(D) No output

Problem 163 ***[MCQ]*** *What is the output?*

```
i = 3
while i > 0:
    i -= 1
else:
    print("while done")
```

(A) No output

(B) `while done`

(C) `while done` is printed three times

(D) `SyntaxError: while` does not support `else`

Problem 164 ***[MCQ]*** *What is the output?*

```
i = 5
while i > 0:
    i -= 1
    if i == 2:
        break
else:
    print("clean exit")
print("after loop")
```

(A) `clean exit` then `after loop`

(B) `after loop`

(C) `clean exit`

(D) No output

Problem 165 ***[MCQ]*** *What is the output?*

```
keys = ["a", "b", "c"]
vals = [1, 2, 3]
d = dict(zip(keys, vals))
print(d)
```

(A) `[('a', 1), ('b', 2), ('c', 3)]`

(B) `{'a': 1, 'b': 2, 'c': 3}`

(C) `{('a', 1): ('b', 2)}`

(D) `TypeError`

Problem 166 ***[MSQ]*** *Which of the following correctly describe Python's* `range` *object?*

(A) `range` *is a lazy sequence; it does not store all values in memory.*

(B) `range(10)` *has 10 elements: 0 through 9.*

(C) `range(1, 5)` *returns a list* `[1, 2, 3, 4]`*.*

(D) `range` *supports negative steps.*

Problem 167 ***[NAT]*** *What is the output of the following code?*

```
count = 0
for i in range(1, 20):
    if i % 2 != 0 and i % 3 != 0:
        count += 1
print(count)
```

Problem 168 ***[MCQ]*** *What is the output?*

```
x = 10
while x > 0:
    x //= 3
    print(x, end=" ")
```

(A) 3 1 0

(B) 3 1

(C) 10 3 1

(D) Infinite loop

Problem 169 ***[MCQ]*** *What is the output?*

```
for i in range(3):
    for j in range(3):
        if i == j:
            break
    else:
        print(f"j={j}", end=" ")
```

(A) j=2 j=2 j=2

(B) No output

(C) j=0 j=1 j=2

(D) j=2

Problem 170 ***[MCQ]*** *What is the output?*

```
s = "gate"
out = ""
for i, c in enumerate(s):
    out += c.upper() if i % 2 == 0 else c
print(out)
```

(A) GATE

(B) gate

(C) GaTe

(D) gAtE

Problem 171 ***[MSQ]*** *Which of the following are* ***correct*** *about the `for...else` construct in Python?*

(A) The `else` block runs if the loop finishes without hitting a `break`.

(B) The `else` block always runs after any `for` loop.

(C) The `else` block is skipped if `break` is executed.

(D) The `else` block also applies to `while` loops.

(E) The `else` block runs even if the iterable is empty.

Problem 172 ***[MCQ]*** *What is the output?*

```
for _ in range(3):
    pass
print(_)
```

(A) NameError

(B) None

(C) 2

(D) 3

Problem 173 ***[MCQ]*** *What is the output?*

```
n = 12
flag = False
for i in range(2, n):
    if n % i == 0:
        flag = True
        break
print("composite" if flag else "prime")
```

(A) prime

(B) composite

(C) True

(D) False

Problem 174 ***[MCQ]*** *What is the output?*

```
for i in range(5):
    if i == 2:
        continue
    if i == 4:
        break
else:
    print("else")
print("end")
```

(A) else then end

(B) end

(C) else

(D) No output

3.4 YouTube Links and QR Codes

Lecture	Details	YouTube Link	QR Code
12	Conditional Constructs in Python	https://youtu.be/VO3tm_fqvts	
13	Loops in Python	https://youtu.be/lowczoWyDMg	
14	Problem Solving on Loops	https://youtu.be/M5DRn3qAkP4	

Chapter 4

Functions

4.1 Defining & Calling Functions

4.1.1 Basic Syntax

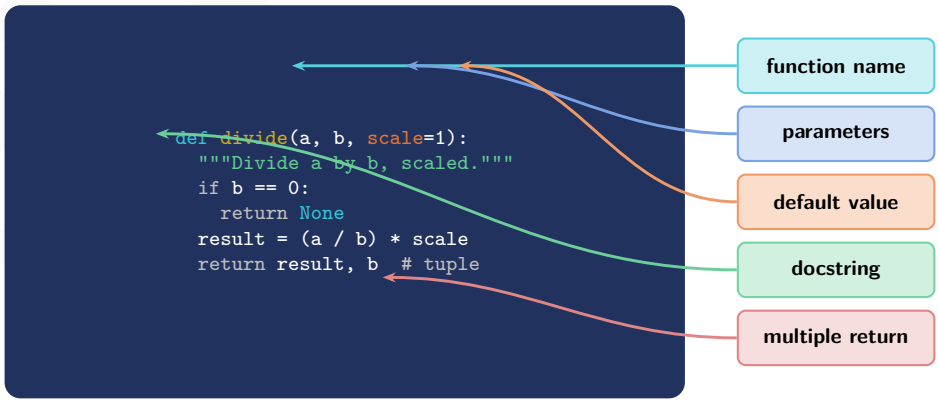
4.1.1.1 def, return, docstrings

Function Definition — Formal Syntax

A Python function is defined with the `def` keyword:

$$\texttt{def}\ \langle \textit{name} \rangle (\langle \textit{params} \rangle):\quad \langle \textit{body} \rangle$$

- `return` exits the function and optionally passes a value back to the caller
- **No `return`** (or bare `return`): function returns `None` implicitly
- **Multiple return values**: Python packs them into a `tuple` (`return x, y` ≡ `return (x, y)`)
- **Docstring**: first statement in the body, if a string literal, is stored as `func.__doc__`; shown by `help(func)`



return Semantics

- **Bare `return` or fall-off-end**: returns `None`
- **`return expr`**: evaluates `expr` and returns it

- **`return a, b, c`**: packs into tuple `(a, b, c)`; caller can unpack: `x, y, z = f()`
- **Multiple `return` statements**: first one reached exits the function
- `return` inside a generator becomes `StopIteration` (covered in generators chapter)

Example 99: Function Definitions — return, Docstrings, Multiple Values

```
# Basic function
def greet(name):
    """Return a greeting string for the given name."""
    return f"Hello, {name}!"

print(greet("GATE"))          # Hello, GATE!
print(greet.__doc__)          # Return a greeting string for the given name.

# No return -> None
def no_return(x):
    x * 2   # result not returned

print(no_return(5))            # None

# Multiple return values (tuple)
def divmod_custom(a, b):
    """Return (quotient, remainder) of a    b."""
    return a // b, a % b       # packs into tuple

q, r = divmod_custom(17, 5)   # unpack
print(q, r)                    # 3  2

result = divmod_custom(17, 5)  # as tuple
print(result)                  # (3, 2)
print(type(result))            # <class 'tuple'>

# Early return
def absolute(n):
    if n >= 0:
        return n
    return -n                  # unreachable if n >= 0

# Docstring access
def area(r):
    """Compute the area of a circle with radius r."""
    import math
    return math.pi * r ** 2

help(area)    # shows the docstring
print(area.__doc__)
```

4.1.2 Function Objects

4.1.2.1 First-Class Functions

Functions as First-Class Objects

In Python, **functions are objects** (instances of `function` type). A **first-class object** can be:

- **Assigned to a variable**: `f = my_func`
- **Passed as an argument**: `map(str, [1,2,3])`
- **Returned from a function**: decorators, closures
- **Stored in a data structure**: `ops = [add, sub, mul]`

- **Has attributes**: `f.__name__`, `f.__doc__`, `f.__code__`

`callable(obj)` returns `True` if `obj` can be called with `()`: functions, classes, objects with `__call__`, built-ins.

Example 100: Functions are Objects — Assign, Pass, Return, Store

```python
def square(n): return n ** 2
def cube(n):   return n ** 3
def negate(n): return -n

# Assign to variable
f = square
print(f(5))            # 25
print(f.__name__)      # square (name stays "square")

# Pass as argument
data = [1, 2, 3, 4, 5]
print(list(map(square, data)))      # [1, 4, 9, 16, 25]
print(list(filter(lambda x: x%2==0, data)))  # [2, 4]

# Store in data structure
ops = {"sq": square, "cb": cube, "neg": negate}
for name, op in ops.items():
    print(f"{name}(3) = {op(3)}")
# sq(3) = 9 / cb(3) = 27 / neg(3) = -3

# Return from function (higher-order)
def make_power(exp):
    def power(base):
        return base ** exp
    return power         # return function object

square2 = make_power(2)
cube2   = make_power(3)
print(square2(5))       # 25
print(cube2(4))         # 64

# callable()
print(callable(square))     # True
print(callable(42))         # False
print(callable(print))      # True
print(callable([]))         # False

class Adder:
    def __call__(self, x): return x + 1

add1 = Adder()
print(callable(add1))  # True  (has __call__)
print(add1(10))         # 11
```

4.2 Arguments & Parameters

4.2.1 Positional and Keyword Arguments

4.2.1.1 Matching Rules

Positional vs Keyword Arguments

Type	Syntax in call	Matching
Positional	`f(1, 2, 3)`	Left-to-right by position
Keyword	`f(b=2, a=1)`	By name, any order
Mixed	`f(1, b=2)`	Positional first, keyword after

Rule: all positional arguments must come *before* keyword arguments in any function call. `f(1, b=2)` is valid; `f(a=1, 2)` is a `SyntaxError`.

4.2.2 Default Parameters

4.2.2.1 Evaluated Once at Definition Time

Default Parameters — Definition-Time Evaluation

Default parameter values are **evaluated exactly once** at the time the `def` statement is executed, *not* at call time.

Consequence: if the default is an *immutable* object (`int`, `str`, `None`), re-evaluation doesn't matter. If it is a *mutable* object (`list`, `dict`), the same object is shared across *all calls* that use the default.

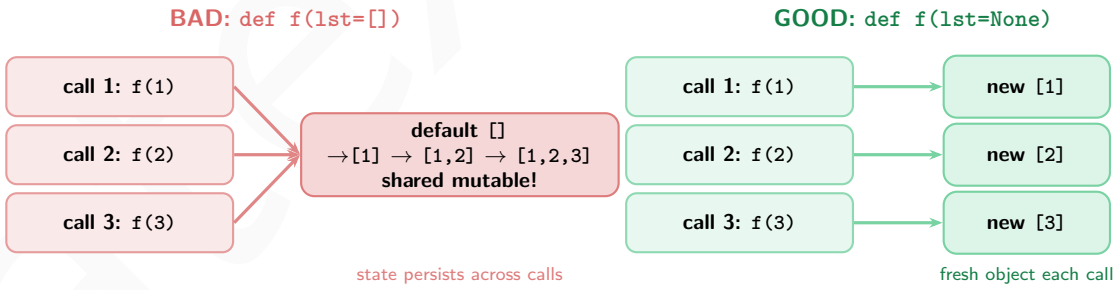
4.2.3 Mutable Default Argument Trap

4.2.3.1 GATE Critical — Shared Mutable Default

Mutable Default Argument Trap

When a mutable object (`list`, `dict`, `set`) is used as a default parameter, it is created **once at function definition time** and **shared across all calls** that rely on the default. State accumulated in one call persists into the next.

Fix: use `None` as the sentinel default and create the mutable object *inside* the function body.



Example 101: Mutable Default — Bug, Fix, and GATE Variants

```
#         BUG: shared mutable default

def append_to(val, lst=[]):     # lst created ONCE at def time
    lst.append(val)
    return lst

print(append_to(1))    # [1]
print(append_to(2))    # [1, 2]   UNEXPECTED!
print(append_to(3))    # [1, 2, 3]  state persists!

# Proof: it's always the same list object
print(id(append_to.__defaults__[0]))    # same id each call
```

```python

#         FIX: use None sentinel


def append_to_fixed(val, lst=None):
    if lst is None:
        lst = []                # new list created every call
    lst.append(val)
    return lst

print(append_to_fixed(1))    # [1]
print(append_to_fixed(2))    # [2]    correct
print(append_to_fixed(3))    # [3]    correct

# Explicit argument bypasses the shared default entirely
print(append_to(4, []))     # [4]  -- no shared state

#         GATE variant: dict default


def add_entry(key, val, store={}):
    store[key] = val
    return store

print(add_entry("a", 1))    # {'a': 1}
print(add_entry("b", 2))    # {'a': 1, 'b': 2}  shared!

#         GATE question: what does this print?

def mystery(x, result=[]):
    result.append(x * 2)
    return result

print(mystery(1))    # [2]
print(mystery(2))    # [2, 4]
print(mystery(3))    # [2, 4, 6]
# Each call appends to the SAME list
```

4.2.4 *args and **kwargs

4.2.4.1 Variable-Length Arguments

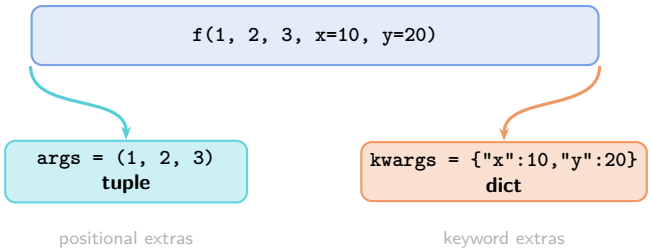
***args and **kwargs — Variable-Length Parameters**

Parameter	Collects	Type
`*args`	Extra positional arguments	`tuple`
`**kwargs`	Extra keyword arguments	`dict`

Parameter order (formal rule):

$$\underbrace{\texttt{pos}}_{\text{positional}} \rightarrow \underbrace{\texttt{*args}}_{\text{var-pos}} \rightarrow \underbrace{\texttt{kw-only}}_{\text{after *}} \rightarrow \underbrace{\texttt{**kwargs}}_{\text{var-kw}}$$

Names `args`/`kwargs` are convention, not syntax — `*nums` and `**opts` work equally.



Keyword-Only and Positional-Only Parameters

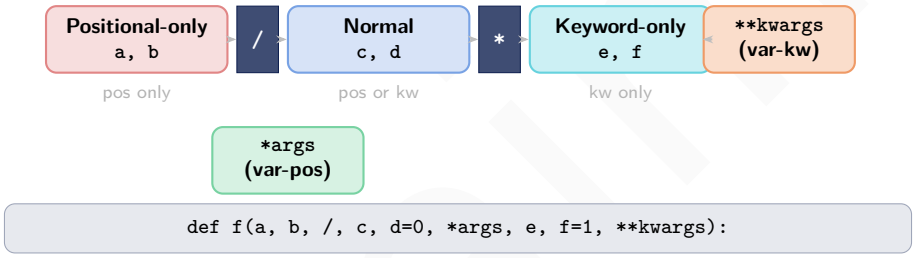
- **Keyword-only** (after `*` or `*args`): parameter *must* be passed by name; cannot be satisfied positionally

  `def f(a, *, b, c=0):` ⇒ `f(1, b=2)` ok; `f(1, 2)` TypeError

- **Positional-only** (before `/`, Python 3.8+): parameter *can only* be passed positionally; `f(a=1)` raises TypeError

  `def f(a, b, /, c):` ⇒ `f(1, 2, c=3)` ok; `f(a=1)` TypeError

- **Combined**: `def f(a, b, /, c, *, d, e):` — `a,b` pos-only; `c` normal; `d,e` kw-only



Example 102: *args and **kwargs — Definitions and Calls

```python
# *args: collects extra positionals into a tuple
def total(*args):
    print(type(args), args)   # <class 'tuple'>
    return sum(args)

print(total(1, 2, 3))         # 6
print(total(10, 20))          # 30
print(total())                # 0  (empty tuple)

# **kwargs: collects extra keywords into a dict
def show_info(**kwargs):
    print(type(kwargs), kwargs)
    for key, val in kwargs.items():
        print(f"  {key}: {val}")

show_info(name="Alice", age=25, city="Delhi")
# <class 'dict'> {'name': 'Alice', 'age': 25, 'city': 'Delhi'}
# name: Alice  / age: 25  / city: Delhi

# Combined: positional + *args + kw-only + **kwargs
def full_sig(a, b, *args, flag=False, **kwargs):
    print(f"a={a}, b={b}")
    print(f"args={args}")         # tuple of extra positionals
    print(f"flag={flag}")         # keyword-only
    print(f"kwargs={kwargs}")     # dict of extra keywords

full_sig(1, 2, 3, 4, 5, flag=True, x=10, y=20)
# a=1, b=2
# args=(3, 4, 5)
# flag=True
# kwargs={'x': 10, 'y': 20}

```

```python
# Forwarding all arguments (common in decorators)
def debug(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args}, {kwargs}")
        result = func(*args, **kwargs)
        print(f"Returned: {result}")
        return result
    return wrapper

@debug
def add(x, y): return x + y
add(3, y=4)
# Calling add with (3,), {'y': 4}
# Returned: 7
```

Example 103: Keyword-Only and Positional-Only Parameters

```python
# Keyword-only (after *)
def connect(host, port, *, timeout=30, retries=3):
    print(f"Connect to {host}:{port}, timeout={timeout}, retries={retries}")

connect("localhost", 8080)                      # defaults
connect("localhost", 8080, timeout=10)          # kw only
connect("localhost", 8080, timeout=5, retries=1)

# TypeError: timeout cannot be passed positionally
try:
    connect("localhost", 8080, 10)
except TypeError as e:
    print(e)  # takes 2 positional arguments but 3 were given

# Positional-only (before /)
def power(base, exp, /):
    return base ** exp

print(power(2, 10))     # 1024
try:
    power(base=2, exp=10)   # TypeError
except TypeError as e:
    print(e)   # got some positional-only arguments passed as keyword

# Combined: pos-only / normal * kw-only
def f(a, b, /, c, *, d):
    return a + b + c + d

print(f(1, 2, 3, d=4))      # 10
print(f(1, 2, c=3, d=4))    # 10  (c can be positional or keyword)
try:
    f(a=1, b=2, c=3, d=4)   # TypeError: a,b are positional-only
except TypeError as e:
    print(e)
```

4.2.5 Argument Unpacking in Calls

4.2.5.1 * and ** at the Call Site

Argument Unpacking — * and ** in Calls

The * and ** operators can appear at the *call site* (not just in function definitions) to *unpack* containers into arguments:

Syntax	Unpacks	Becomes
`f(*seq)`	list/tuple/iterable	positional arguments
`f(**dct)`	dict	keyword arguments

Multiple unpacking is allowed in a single call (Python 3.5+): `f(*a, *b, **c, **d)`

Example 104: Unpacking at Call Site

```
def add3(x, y, z):
    return x + y + z

args = (3, 4, 5)
print(add3(*args))          # 12  same as add3(3, 4, 5)

vals = [10, 20, 30]
print(add3(*vals))          # 60

# ** unpacks dict as keyword args
def describe(name, age, city):
    return f"{name}, {age}, from {city}"

info = {"age": 25, "city": "Delhi"}
print(describe("Alice", **info))   # Alice, 25, from Delhi

# Full unpack: *args + **kwargs
def f(a, b, c, x=0, y=0):
    return a + b + c + x + y

pos  = [1, 2, 3]
kw   = {"x": 10, "y": 20}
print(f(*pos, **kw))   # 36

# Multiple unpacking (Python 3.5+)
a = [1, 2]
b = [3, 4]
print([*a, *b])             # [1, 2, 3, 4]  (list merge)
print({**{"a":1}, **{"b":2}})  # {'a':1,'b':2}  (dict merge)

# Practical: pass config dict to function
import json
config = {"indent": 2, "sort_keys": True}
data = {"b": 2, "a": 1}
print(json.dumps(data, **config))
# {"a": 1, "b": 2}  (sorted, indented)
```

4.3 Pass-by-Object-Reference

4.3.0.1 How Python Passes Arguments

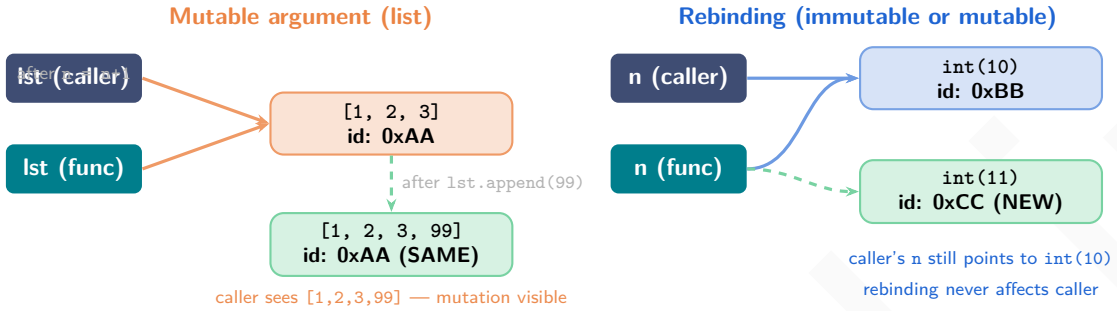
Pass-by-Object-Reference (Call by Sharing)

Python uses **pass-by-object-reference** (also called *call by sharing* or *call by assignment*). The function receives a **reference to the same object** — not a copy of the object, not the variable itself.

Key rules:

1. **Rebinding** (`x = new_val`) inside a function creates a *new local binding*; it **never** affects the caller's variable
2. **Mutation** of a mutable object (e.g. `lst.append(x)`) *is visible* to the caller because both names refer to the same object

3. **Immutable objects** (int, str, tuple) cannot be mutated — any "modification" creates a new object; caller is unaffected



Pass-by-Object-Reference Rules Summary

- **Mutation (mutable objects)**: `lst.append(x)`, `dct[k] = v`, `obj.attr = val` — caller *sees the change*
- **Rebinding**: `x = something_new` inside function — caller's name *unaffected*; local name now refers to a different object
- **Immutable types** (int, str, float, tuple, frozenset): impossible to mutate in place — any "modification" produces a new object (rebinding)
- **Augmented assignment on mutable**: `lst += [x]` calls `__iadd__` which mutates in place and rebinds — caller *sees change*
- **Augmented assignment on immutable**: `n += 1` creates new object and rebinds local name — caller *unaffected*
- **To prevent mutation**: pass `lst[:]`, `tuple(lst)`, or `copy.deepcopy(obj)`

Example 105: Mutation vs Rebinding — Detailed

```
#         1. Mutable argument: mutation IS visible

def add_element(lst, val):
    lst.append(val)         # mutates the object caller passed

my_list = [1, 2, 3]
add_element(my_list, 99)
print(my_list)              # [1, 2, 3, 99]  -- changed!

#         2. Rebinding does NOT affect caller

def try_rebind(lst):
    lst = [10, 20, 30]      # local rebind; caller unaffected
    lst.append(40)

my_list2 = [1, 2, 3]
try_rebind(my_list2)
print(my_list2)             # [1, 2, 3]  -- unchanged!

#         3. Immutable: always creates new object

def increment(n):
    n += 1                  # creates new int; local rebind
    print(f"inside: {n}")

x = 10
increment(x)                # inside: 11
print(x)                    # 10  -- unchanged!

#         4. Same with strings
```

```python
def modify_str(s):
    s += " World"           # new str object
    print(f"inside: {s}")

greeting = "Hello"
modify_str(greeting)        # inside: Hello World
print(greeting)             # Hello  -- unchanged!

#        5. Augmented += on mutable: mutates AND rebinds
def extend_list(lst):
    lst += [99]             # calls __iadd__: mutates in place!

L = [1, 2, 3]
extend_list(L)
print(L)                    # [1, 2, 3, 99]  -- changed!

#        6. How to truly prevent mutation: pass a copy
import copy

def safe_process(lst):
    lst.append(999)         # modifies the copy, not original
    return lst

original = [1, 2, 3]
result = safe_process(original[:])      # shallow copy
print(original)             # [1, 2, 3]  -- safe!

result2 = safe_process(copy.deepcopy(original))
print(original)             # [1, 2, 3]  -- safe!
```

Example 106: Tricky Function Questions

```python
# Q1: Trace the output
def f(x, lst=[]):
    lst.append(x)
    return lst

print(f(1))     # [1]
print(f(2))     # [1, 2]   shared default!
print(f(3,[]))  # [3]    explicit arg bypasses default
print(f(4))     # [1, 2, 4]   still using shared default

# Q2: What does this print?
def swap(a, b):
    a, b = b, a             # local rebind; tuple unpacking
    print(f"inside: {a}, {b}")

x, y = 10, 20
swap(x, y)                  # inside: 20, 10
print(x, y)                 # 10 20  -- caller unchanged!

# Q3: Mutable inside immutable (tuple containing list)
def modify(t):
    t[0].append(99)         # modifies the LIST inside the tuple
    # t[0] = []             # would TypeError: tuple is immutable

data = ([1, 2], "hello")
modify(data)
print(data)                 # ([1, 2, 99], 'hello')  list inside changed!

# Q4: *args receives a tuple
def check_types(*args):
    print(type(args))       # <class 'tuple'>
    for a in args:
        print(type(a).__name__, a)

```

```python
check_types(1, "hi", [1,2], None)
# tuple
# int 1
# str hi
# list [1, 2]
# NoneType None

# Q5: Keyword-only enforcement
def strict(a, b, *, c):
    return a + b + c

print(strict(1, 2, c=3))   # 6
try:
    strict(1, 2, 3)          # TypeError: c is keyword-only
except TypeError as e:
    print(e)
```

4.4 Scope — LEGB Rule

4.4.1 The Four Scopes

4.4.1.1 LEGB Lookup Order

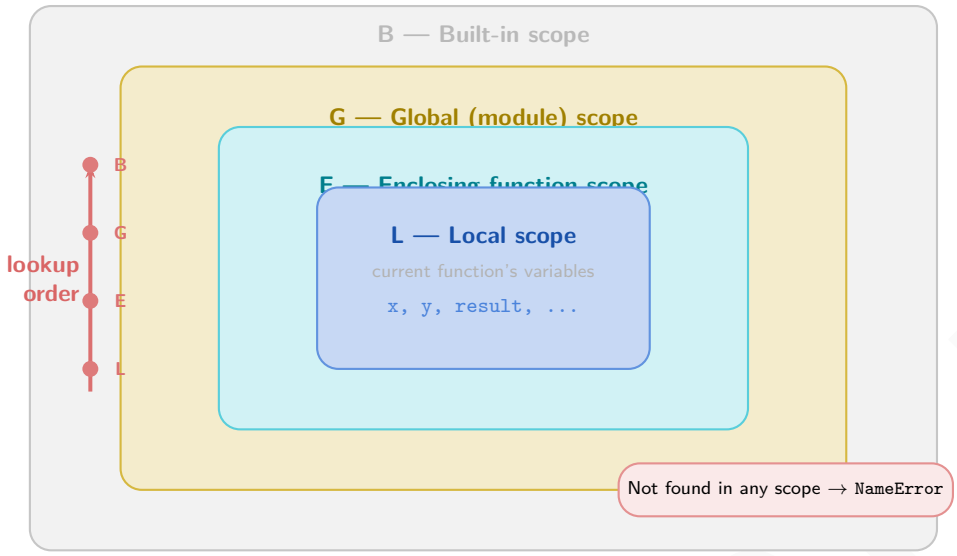
Scope and the LEGB Rule

Scope defines where in code a name (variable, function, class) is visible and accessible. Python resolves names using the **LEGB rule** — a strict lookup order through four nested scopes:

Letter	Scope	Where
L	Local	Inside the current function body
E	Enclosing	Any enclosing function(s), inner-to-outer
G	Global	Module-level namespace
B	Built-in	`builtins` module (`len`, `print`, ...)

Python searches **L** → **E** → **G** → **B** in order and uses the *first* match found. If no match exists in any scope, a `NameError` is raised.

Assignment creates a local binding unless `global` or `nonlocal` is declared. Reading a name never creates a binding.



Scope Creation Rules

- **New scope created by**: `def`, `class`, `lambda` (not by `if`, `for`, `while`, `with` — these share the enclosing scope)
- **Assignment in a function** (=, +=, `for` target, `with ... as`, function parameter, `import`) creates a *local* variable in that function's scope
- **Reading only** (no assignment anywhere in the function): looks up LEGB
- **UnboundLocalError**: Python sees an assignment to `x` anywhere in a function, so it treats `x` as local — reading `x` before the assignment raises `UnboundLocalError`, not `NameError`
- **Inspect scopes**: `locals()`, `globals()`, `vars()`, `dir()`

Example 107: LEGB Lookup — Step by Step

```
# Built-in scope
print(len([1,2,3]))     # len found in B (built-in)

# Global scope
x = "global"
def show_global():
    print(x)            # not in L or E; found in G
show_global()           # global

# Local shadows global
x = "global"
def show_local():
    x = "local"         # creates LOCAL binding
    print(x)            # found in L
show_local()            # local
print(x)                # global  (global unchanged)

# LEGB full chain
PI = 3.14159            # G

def outer():
    base = 10           # E (for inner)

    def inner():
        radius = 5      # L
        area = PI * radius ** 2     # radius:L, PI:G
        result = base * radius      # base:E
        print(f"area={area:.2f}, result={result}")

```

```
    inner()

outer()   # area=78.54, result=50

# if/for do NOT create new scope
for i in range(3):
    loop_var = i * 2
print(loop_var)   # 4  (loop_var leaks into enclosing scope!)
# this would be an error in Java/C++ -- not in Python

# UnboundLocalError trap
val = 100
def tricky():
    print(val)    # ERROR: Python sees 'val = ...' below,
    val = 200     # so 'val' is treated as LOCAL throughout
# tricky()       # UnboundLocalError: free variable 'val' ...
```

Example 108: locals(), globals(), vars() Introspection

```
MODULE_VAR = 42

def demo(a, b):
    c = a + b
    d = "local"
    print(locals())   # {'a': 10, 'b': 20, 'c': 30, 'd': 'local'}

demo(10, 20)

# globals() returns the module-level namespace dict
g = globals()
print("MODULE_VAR" in g)   # True
print(g["MODULE_VAR"])     # 42

# Dynamic lookup using globals()
def call_by_name(func_name, *args):
    func = globals()[func_name]   # look up function by name string
    return func(*args)

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

print(call_by_name("add", 3, 4))   # 7
print(call_by_name("mul", 3, 4))   # 12
```

4.4.2 global Keyword

4.4.2.1 Accessing and Modifying Module-Level Names

global Keyword — Definition

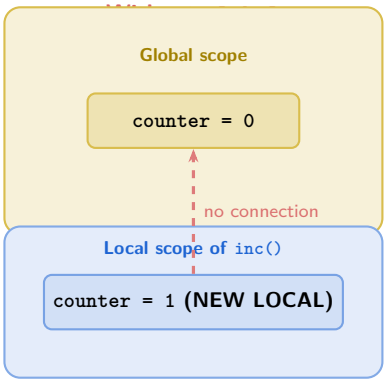
By default, *assigning* to a name inside a function creates a **local** variable. To make an assignment target the **module-level** (global) name instead, declare it with `global`:

`global` ⟨*name*⟩

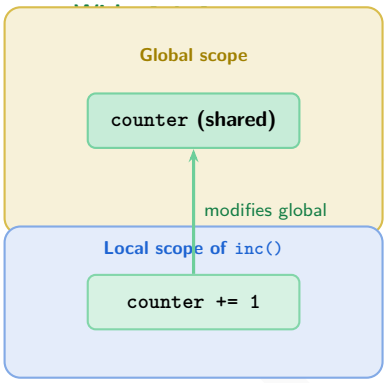
Rules:

- `global x` must appear *before* any use of `x` in the function (best practice: at the very top of the function)
- After `global x`, *all* reads and writes in that function refer to the global `x`
- **Reading** a global without assignment does *not* need `global` — LEGB handles it automatically
- Can declare multiple: `global x, y, z`

- **Anti-pattern**: excessive `global` use makes code hard to test and reason about; prefer returning values



global counter stays 0



global counter becomes 1, 2, 3...

Example 109: global — Counter, Registry, and the No-global Reading Rule

```python
#  global needed for ASSIGNMENT
counter = 0

def increment():
    global counter          # declare intent to modify global
    counter += 1

increment(); increment(); increment()
print(counter)              # 3

#        Without global: creates local, global untouched
total = 100
def try_modify():
    total = 200             # creates LOCAL 'total'; global unchanged
    print(f"inside: {total}")

try_modify()                # inside: 200
print(total)                # 100  -- global unchanged!

#        Reading global WITHOUT global keyword (no declaration needed)
PI = 3.14159
def area(r):
    return PI * r ** 2      # reads PI from G via LEGB -- no 'global' needed

print(area(5))              # 78.53975

#        UnboundLocalError: assignment makes var local throughout
value = 50
def broken():
    print(value)            # UnboundLocalError! assignment below
    value = 99              # Python treats 'value' as local in whole func

# Fix with global:
def fixed():
    global value
    print(value)            # 50
    value = 99
fixed()
print(value)                # 99  (global modified)

#        Practical: function registry

_registry = {}

def register(func):
    global _registry
    _registry[func.__name__] = func
```

```
    return func

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

@register
def mul(a, b): return a * b

print(_registry)      # {'add': <function add...>, 'mul': <function mul...>}
print(_registry["add"](3, 4))    # 7
```

4.4.3 nonlocal Keyword

4.4.3.1 Rebinding in Enclosing Scope (Closures)

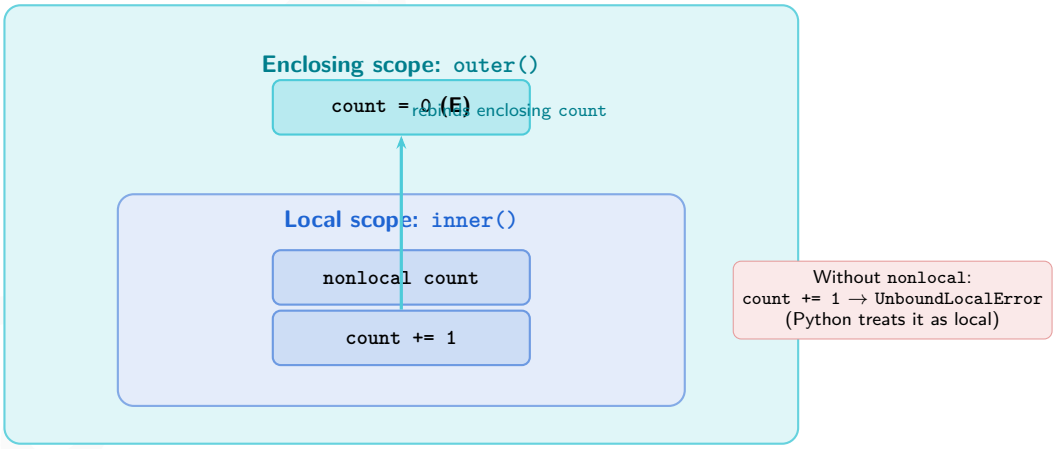
nonlocal Keyword — Definition

`nonlocal` rebinds a name in the **nearest enclosing (non-global) function scope**. It is needed *only for assignment*; reading an enclosing variable works automatically via LEGB.

`nonlocal` ⟨*name*⟩

Rules:

- The name must *already exist* in an enclosing function scope (not module-level) — `nonlocal x` when `x` is only global raises `SyntaxError`
- After declaration, all reads *and* writes refer to the enclosing variable
- **Mutation of a mutable object does NOT require `nonlocal`**: `lst.append(x)` mutates the object the enclosing name refers to; it never rebinds the name
- Chains: if multiple levels nest, `nonlocal` reaches the *nearest* enclosing scope that has the name



Action	Needs `global`?	Needs `nonlocal`?
Read global variable	No	No
Assign to global	Yes	N/A
Read enclosing variable	No	No
Rebind enclosing variable	N/A	Yes
Mutate enclosing mutable	No	No

Example 110: nonlocal — Counter Closure and State Mutation

```
#         nonlocal needed for rebinding

def make_counter(start=0):
    count = start               # E scope

    def increment(step=1):
        nonlocal count          # declare: modify E scope's 'count'
        count += step           # rebind enclosing count
        return count

    def reset():
        nonlocal count
        count = start

    def get():
        return count            # READ only -- no nonlocal needed

    return increment, reset, get

inc, rst, get = make_counter(0)
print(inc())       # 1
print(inc())       # 2
print(inc(5))      # 7
print(get())       # 7
rst()
print(get())       # 0  (reset to start)

#         WITHOUT nonlocal: reading works, rebinding fails
def outer():
    x = 10

    def inner_read():
        print(x)        # OK: LEGB reads from E

    def inner_write():
        x += 1          # UnboundLocalError! treated as local

    inner_read()        # 10

outer()
# inner_write() would crash with UnboundLocalError

#         Mutation of mutable does NOT need nonlocal

def make_history():
    log = []                    # E scope, mutable

    def record(event):
        log.append(event)       # MUTATES the list; no nonlocal needed
        return log

    def clear():
        log.clear()             # also mutation; no nonlocal needed
```

```
    return record, clear

rec, clr = make_history()
print(rec("login"))    # ['login']
print(rec("query"))    # ['login', 'query']
clr()
print(rec("logout"))   # ['logout']   (fresh after clear)

#        nonlocal chains multiple levels

def level1():
    n = 0

    def level2():

        def level3():
            nonlocal n          # skips level2; reaches level1's n
            n += 100

        level3()
        print(f"after level3: {n}")    # 100

    level2()
    print(f"level1 n: {n}")            # 100

level1()
```

Example 111: Closures — State via Enclosing Scope

```
# Closure: inner function "closes over" enclosing variables
def make_multiplier(factor):
    """Return a function that multiplies its argument by factor."""
    def multiply(x):
        return x * factor       # factor captured from E scope
    return multiply

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))    # 10
print(triple(5))    # 15
print(double.__closure__)   # (<cell at 0x...>,)
print(double.__closure__[0].cell_contents)  # 2

# Closure cell shared across multiple inner functions
def make_adder_pair(n):
    def add(x):
        nonlocal n
        n += x
        return n
    def subtract(x):
        nonlocal n
        n -= x
        return n
    return add, subtract

add5, sub5 = make_adder_pair(100)
print(add5(10))     # 110
print(sub5(5))      # 105  (shared n)
print(add5(20))     # 125

# GOTCHA: closure captures variable, not value
def make_funcs_bad():
    funcs = []
    for i in range(3):
        funcs.append(lambda: i)    # ALL capture the SAME 'i'
    return funcs
```

```
fs = make_funcs_bad()
print([f() for f in fs])    # [2, 2, 2]  -- all see final i=2!

# FIX: capture by default argument (snapshot the value)
def make_funcs_good():
    funcs = []
    for i in range(3):
        funcs.append(lambda i=i: i)    # default arg binds current i
    return funcs

fs2 = make_funcs_good()
print([f() for f in fs2])  # [0, 1, 2]  -- correct!
```

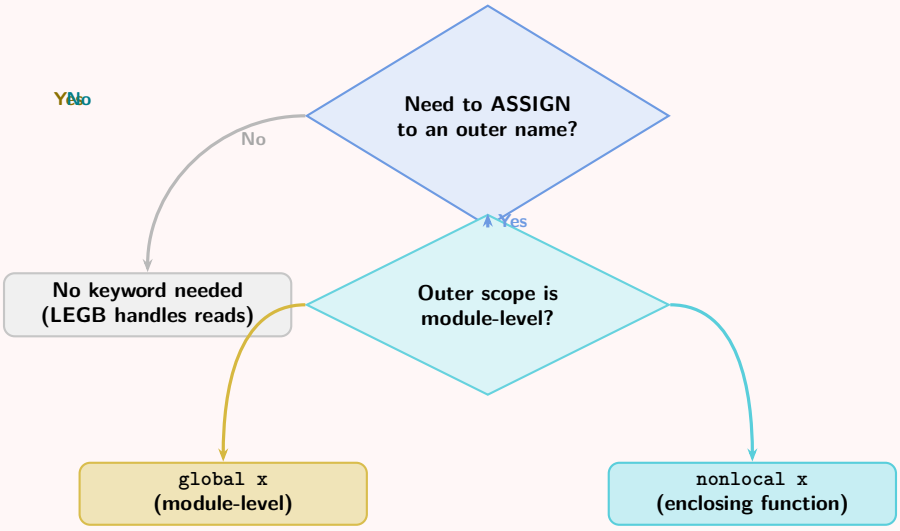
Example 112: Scope Tracing Questions

```
# Q1: What does this print?
x = 1
def f():
    x = 2
    def g():
        x = 3
        print(x)    # L: 3
    g()
    print(x)        # L (f's): 2
f()
print(x)            # G: 1
# Output: 3 / 2 / 1

# Q2: global vs local -- UnboundLocalError
y = 10
def h():
    y += 5          # UnboundLocalError: y treated as local
# h()              # would crash

def h_fixed():
    global y
    y += 5

h_fixed()
print(y)            # 15

# Q3: nonlocal or mutation?
def outer():
    data = []

    def add_item(x):
        data.append(x)    # mutation: no nonlocal needed
        # data = [x]      # THIS would need nonlocal (rebinding)

    add_item(1); add_item(2); add_item(3)
    print(data)     # [1, 2, 3]

outer()

# Q4: closure variable capture
def make_powers():
    return [lambda x, n=n: x**n for n in range(1, 4)]

square, cube_fn, fourth = make_powers()
print(square(3))     # 3   (3^1)
print(cube_fn(3))    # 9   (3^2)
print(fourth(3))     # 27  (3^3)

# Q5: scope chain
a = "global_a"
def outer2():
    b = "enclosing_b"
```

```
    def inner():
        c = "local_c"
        print(a, b, c)    # G, E, L -- all accessible

    inner()

outer2()    # global_a  enclosing_b  local_c

# Q6: What happens here?
total = 0
def add_to_total(n):
    total = total + n  # UnboundLocalError! (no global declaration)
    return total

# Fix:
def add_to_total_fixed(n):
    global total
    total = total + n
    return total

print(add_to_total_fixed(5))    # 5
print(add_to_total_fixed(3))    # 8
```

4.4.3.2 global vs nonlocal — Quick Reference

global vs nonlocal — Decision Guide



- Use `global` when the target variable lives at *module level*
- Use `nonlocal` when the target variable lives in an *enclosing function* (closure)
- Neither is needed to *read* an outer variable or to *mutate a mutable object* (list, dict, set) in an outer scope
- Prefer **returning values** over `global` whenever possible — cleaner, testable, side-effect-free

4.5 Recursion

4.5.1 Mechanics

4.5.1.1 Call Stack, Base Case, Recursive Case

Recursion — Definition and Mechanics

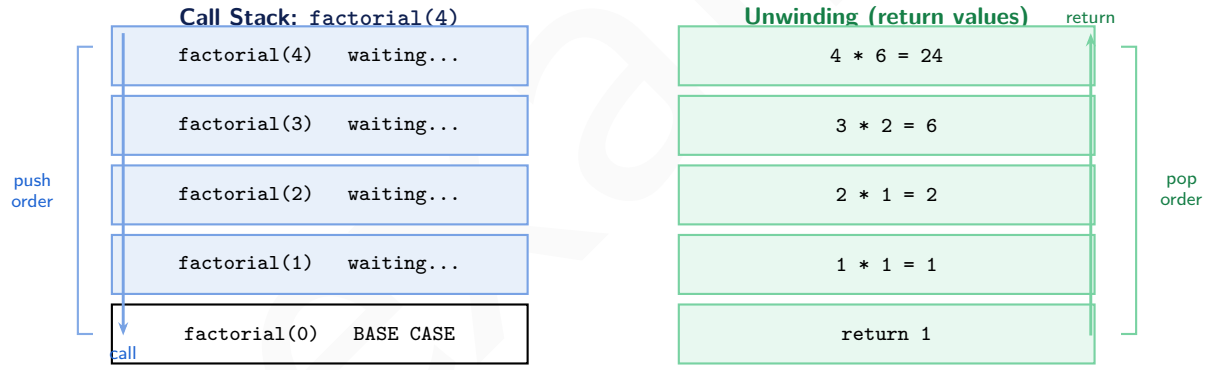
A function is **recursive** when it calls itself (directly or indirectly) as part of its own definition.

Every recursive function needs two components:

1. **Base case**: a condition under which the function returns *directly* without another recursive call — terminates the recursion
2. **Recursive case**: reduces the problem toward the base case and makes a recursive call with the reduced input

Call stack mechanics:

- Each function call pushes a **frame** (local variables, return address) onto the call stack
- When the call returns, its frame is **popped**
- CPython's default recursion limit is **1000** frames; exceeding it raises `RecursionError: maximum recursion depth exceeded`
- Check / change: `sys.getrecursionlimit()`, `sys.setrecursionlimit(n)`



Recursion Properties and Limits

- **Stack depth**: each recursive call adds one frame; depth = number of open calls at any instant
- **Maximum depth**: CPython default is 1000; `sys.setrecursionlimit(10000)` can increase it (use with care)
- **Space complexity**: $O(\text{depth})$ stack frames, each holding local variables
- **RecursionError** most often caused by: missing base case, wrong base case condition, or input that never reaches the base case
- **No tail-call optimisation (TCO)** in Python: even a pure tail-recursive function uses one frame per call (Python intentionally omits TCO)

Example 113: Factorial — Base Case, Recursive Case, RecursionError

```
1 import sys
2
3 def factorial(n):
```

```python
    """n! for non-negative integer n."""
    if n == 0:            # BASE CASE: terminates recursion
        return 1
    return n * factorial(n - 1)    # RECURSIVE CASE: n reduced by 1 each time

print(factorial(0))    # 1
print(factorial(5))    # 120
print(factorial(10))   # 3628800

# Recursion depth
def depth_count(n):
    if n == 0:
        return 0
    return 1 + depth_count(n - 1)

print(depth_count(100))    # 100 frames on the stack at peak

# RecursionError: no base case reached
def bad_recursion(n):
    return bad_recursion(n - 1)    # never reaches 0 for negative n

try:
    bad_recursion(5)
except RecursionError as e:
    print(str(e)[:50])    # maximum recursion depth exceeded

# Check and change limit
print(sys.getrecursionlimit())     # 1000 (default)
# sys.setrecursionlimit(5000)      # increase carefully
```

4.5.2 Tracing Recursive Calls

4.5.2.1 Call Trees, Node Count, Fibonacci

Tracing Recursion — Call Tree Method

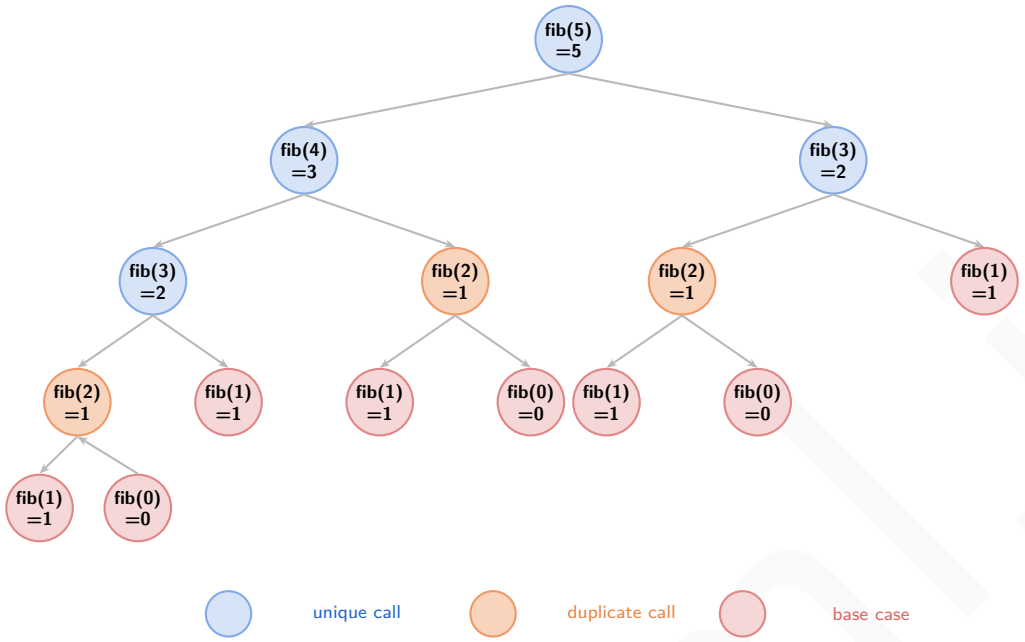
To trace any recursive function:

1. Draw the **call tree**: each node is one function call; children are the recursive calls it makes
2. **Total activations** = total number of nodes (including leaves)
3. Compute return values **bottom-up**: leaves return first; parents combine children's returns

Fibonacci complexity:

`fib(n) = fib(n-1) + fib(n-2)`

- Total calls for `fib(n)`: $T(n) = T(n-1) + T(n-2) + 1 \approx 2^{n/2}$ — **exponential**
- Recomputes the same sub-problems many times (e.g. `fib(2)` computed 3 times inside `fib(5)`)



Total nodes = 15 calls for fib(5)

Call Count Formulas for Common Recursions

Recursion	Total calls	Complexity
`factorial(n)`	$n+1$	$O(n)$
`fib(n)` (naive)	$2 \cdot \texttt{fib}(n+1) - 1 \approx \phi^n$	$O(\phi^n)$
Binary search `search(n)`	$\log_2 n + 1$	$O(\log n)$
Sum of digits `dig(n)`	$\lfloor \log_{10} n \rfloor + 1$	$O(\log n)$
Power `pow(x,n)` (fast)	$2 \log_2 n$	$O(\log n)$
Tower of Hanoi `hanoi(n)`	$2^n - 1$	$O(2^n)$

GATE tip: count nodes, not edges. Each node = one function call = one frame.

Example 114: Tracing Factorial and Fibonacci — Manual Unwinding

```
# Traced factorial(4)
# factorial(4) -> 4 * factorial(3)
#                      3 * factorial(2)
#                               2 * factorial(1)
#                                        1 * factorial(0)
#                                                 = 1   (base)
#                                        = 1*1 = 1
#                               = 2*1 = 2
#                      = 3*2 = 6
# = 4*6 = 24

# Instrument to count calls
call_count = 0

def fib_count(n):
    global call_count
    call_count += 1
    if n <= 1:
```

```
        return n
    return fib_count(n-1) + fib_count(n-2)

for k in range(1, 8):
    call_count = 0
    result = fib_count(k)
    print(f"fib({k}) = {result:3d}  calls = {call_count}")
# fib(1) =   1  calls =   1
# fib(2) =   1  calls =   3
# fib(3) =   2  calls =   5
# fib(4) =   3  calls =   9
# fib(5) =   5  calls =  15
# fib(6) =   8  calls =  25
# fib(7) =  13  calls =  41

# Pattern: calls(n) = calls(n-1) + calls(n-2) + 1
# Roughly doubles every 2 steps -> exponential

# GATE question: how many times is fib(2) called in fib(6)?
# Answer: 3 times (draw the tree to verify)
```

4.5.3 Return Value Aggregation

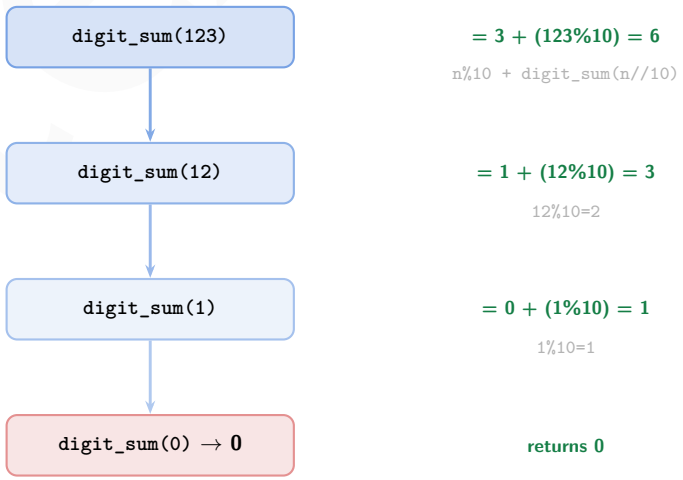
4.5.3.1 Accumulating Results Bottom-Up

Return Value Aggregation Patterns

Recursive functions aggregate results from sub-calls by:

- **Sum**: `return f(n-1) + value`
- **Count**: `return f(n-1) + 1` or `+ (1 if cond else 0)`
- **Max/Min**: `return max(f(left), f(right), current)`
- **List building**: `return [current] + f(rest)`
- **Boolean**: `return cond and f(rest)`

Tracing rule: resolve the deepest call first (base case), then work upward substituting return values.



Example 115: Aggregation — Sum, Count, Flatten, Palindrome

```
# 1. Sum of digits
def digit_sum(n):
```

```
    n = abs(n)
    if n < 10:
        return n
    return n % 10 + digit_sum(n // 10)

print(digit_sum(123))    # 6  (1+2+3)
print(digit_sum(9999))   # 36

# 2. Count occurrences in nested list
def count_in(lst, target):
    if not lst:
        return 0
    head, *tail = lst
    if isinstance(head, list):
        return count_in(head, target) + count_in(tail, target)
    return (1 if head == target else 0) + count_in(tail, target)

data = [1, [2, 1, [3, 1]], 1, [1]]
print(count_in(data, 1))   # 5

# 3. Flatten nested list
def flatten(lst):
    if not lst:
        return []
    head, *tail = lst
    if isinstance(head, list):
        return flatten(head) + flatten(tail)
    return [head] + flatten(tail)

nested = [1, [2, [3, 4]], [5, 6], 7]
print(flatten(nested))   # [1, 2, 3, 4, 5, 6, 7]

# 4. Palindrome check
def is_palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and is_palindrome(s[1:-1])

print(is_palindrome("racecar"))  # True
print(is_palindrome("python"))   # False

# 5. Power (fast exponentiation: O(log n) calls)
def power(base, exp):
    if exp == 0:    return 1
    if exp % 2 == 0:
        half = power(base, exp // 2)
        return half * half        # only ONE recursive call!
    return base * power(base, exp - 1)

print(power(2, 10))   # 1024
print(power(3, 5))    # 243

# 6. Binary search (recursive)
def binary_search(lst, target, lo=0, hi=None):
    if hi is None: hi = len(lst) - 1
    if lo > hi:    return -1
    mid = (lo + hi) // 2
    if lst[mid] == target:   return mid
    if lst[mid] < target:    return binary_search(lst, target, mid+1, hi)
    return binary_search(lst, target, lo, mid-1)

arr = [1,3,5,7,9,11,13,15,17,19]
print(binary_search(arr, 11))   # 5
print(binary_search(arr, 6))    # -1
```

4.5.4 Recursive vs Iterative Equivalence

4.5.4.1 Converting Recursion to Iteration

Recursive ≡ Iterative + Explicit Stack

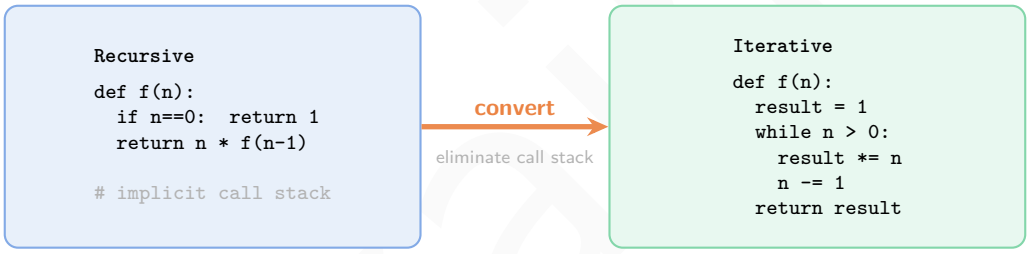
Any recursive algorithm can be converted to an iterative one by:

1. Replacing the call stack with an **explicit stack** (Python `list` used as LIFO)
2. Pushing the "work to do" instead of making a recursive call
3. Popping from the stack and processing until the stack is empty

When to prefer iterative:

- Deep recursion risks `RecursionError` (stack depth > 1000)
- Performance: function call overhead is eliminated
- Python has no tail-call optimisation — even pure tail recursion uses one frame per call

Tail recursion: a recursive call is in *tail position* when it is the *last action* of the function (no further computation after it returns). Python does **not** optimise tail calls — tail-recursive functions still consume $O(n)$ stack frames.



Example 116: Recursive ↔ Iterative — Three Patterns

```python
#         1. Factorial: trivial tail recursion -> loop

def fact_recursive(n):
    if n == 0: return 1
    return n * fact_recursive(n - 1)     # NOT tail: multiply after return

def fact_iterative(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result

print(fact_iterative(10))    # 3628800

#         2. Fibonacci: recursion -> DP / iteration

def fib_iterative(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

print([fib_iterative(k) for k in range(8)])
# [0, 1, 1, 2, 3, 5, 8, 13]

#         3. DFS (tree traversal): recursion -> explicit stack
def dfs_recursive(graph, node, visited=None):
    if visited is None: visited = set()
    visited.add(node)
```

```
    for nb in graph.get(node, []):
        if nb not in visited:
            dfs_recursive(graph, nb, visited)
    return visited

def dfs_iterative(graph, start):
    visited = set()
    stack   = [start]
    while stack:
        node = stack.pop()      # LIFO -- explicit stack replaces call stack
        if node not in visited:
            visited.add(node)
            stack.extend(nb for nb in graph.get(node, [])
                         if nb not in visited)
    return visited

graph = {1:[2,3], 2:[4,5], 3:[6], 4:[], 5:[], 6:[]}
print(dfs_recursive(graph, 1))  # {1, 2, 3, 4, 5, 6}
print(dfs_iterative(graph, 1))  # {1, 2, 3, 4, 5, 6}

#        4. Tail recursion: Python still uses O(n) frames

def fact_tail(n, acc=1):
    if n == 0: return acc
    return fact_tail(n - 1, acc * n)  # tail position: no work after return

# Still O(n) stack frames in Python (no TCO)
# For n=1000, this still RecursionErrors; use the loop version
```

4.5.5 Memoisation

4.5.5.1 Caching with lru_cache and Manual Dicts

Memoisation — Definition

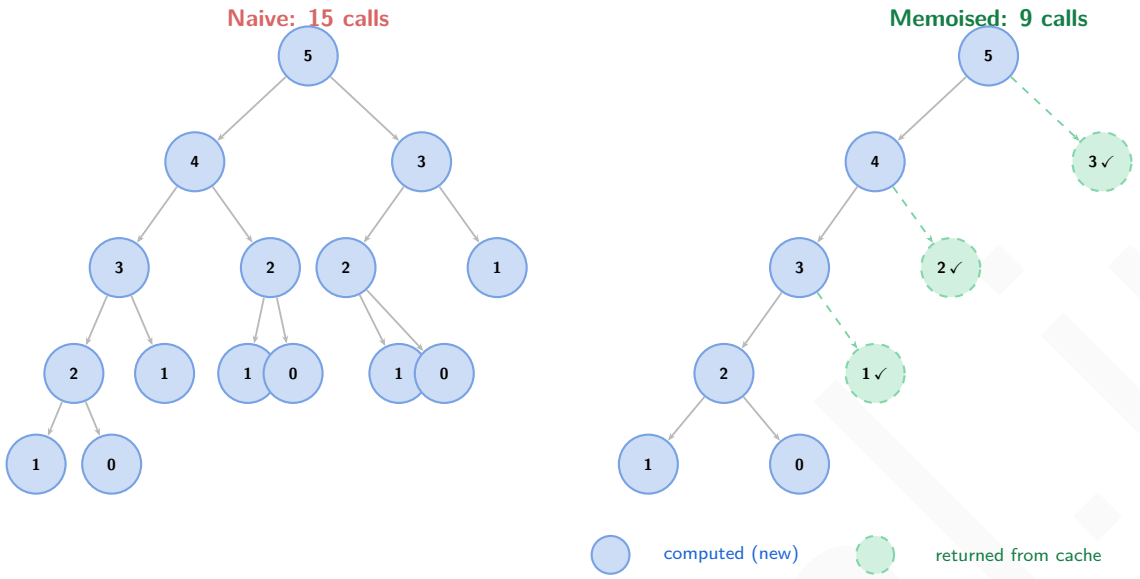
Memoisation (from *memo*, Latin for "to be remembered") is an optimisation that **caches the return values** of a function so that repeated calls with the same arguments are returned instantly from cache instead of recomputed.

Effect on Fibonacci:

Version	Calls for fib(n)	Complexity
Naive recursive	$\approx 2^{n/2}$ (exponential)	$O(\phi^n)$
Memoised	$2n-1$ (each unique subproblem once)	$O(n)$
Iterative DP	$n-1$	$O(n)$

`functools.lru_cache`:

- `@lru_cache(maxsize=None)`: unlimited cache (equivalent to `@cache` in Python 3.9+)
- `@lru_cache(maxsize=128)`: keeps the 128 most recently used; evicts oldest when full (Least Recently Used policy)
- Arguments must be **hashable** (the cache key is the args tuple)
- `func.cache_info()`: `hits, misses, maxsize, currsize`
- `func.cache_clear()`: empties the cache



Example 117: lru_cache — Fibonacci, Coin Change, Climbing Stairs

```python
from functools import lru_cache, cache

#         1. Fibonacci with lru_cache


@lru_cache(maxsize=None)
def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)

print(fib(50))     # 12586269025  (instant with cache)
print(fib(100))    # 354224848179261915075

print(fib.cache_info())
# CacheInfo(hits=97, misses=101, maxsize=None, currsize=101)

fib.cache_clear()    # reset cache

#         2. Manual dict cache (explicit, same effect)

memo = {}
def fib_manual(n):
    if n in memo: return memo[n]
    if n <= 1:    return n
    memo[n] = fib_manual(n-1) + fib_manual(n-2)
    return memo[n]

print(fib_manual(30))  # 832040

#         3. Coin change (number of ways)

@lru_cache(maxsize=None)
def coin_ways(amount, coins=(1, 5, 10, 25)):
    if amount == 0: return 1
    if amount < 0:  return 0
    return sum(coin_ways(amount - c, coins) for c in coins)

print(coin_ways(30))    # number of ways to make 30 cents

#         4. Climbing stairs (1 or 2 steps at a time)

@cache     # Python 3.9+: same as lru_cache(maxsize=None)
def climb(n):
    if n <= 1: return 1
    if n == 2: return 2
```

```
    return climb(n-1) + climb(n-2)

print([climb(k) for k in range(1, 9)])
# [1, 2, 3, 5, 8, 13, 21, 34]  (Fibonacci-like)

#         5. Measuring the speedup


import time

def fib_slow(n):
    if n <= 1: return n
    return fib_slow(n-1) + fib_slow(n-2)

t0 = time.perf_counter()
fib_slow(35)
print(f"Without cache: {time.perf_counter()-t0:.3f}s")    # ~2-5s

t0 = time.perf_counter()
fib(35)
print(f"With cache   : {time.perf_counter()-t0:.6f}s")    # ~0.000001s

#         6. Cache with maxsize (LRU eviction)

@lru_cache(maxsize=4)
def expensive(n):
    print(f"  computing {n}")
    return n * n

expensive(1); expensive(2); expensive(3); expensive(4)
expensive(1)    # HIT (still in cache)
expensive(5)    # MISS -> evicts LRU entry (2)
expensive(2)    # MISS again (was evicted)
print(expensive.cache_info())
```

Example 118: Recursion — Classic GATE Problems

```
#         Tower of Hanoi (exactly 2^n - 1 moves)

def hanoi(n, src="A", dst="C", aux="B"):
    if n == 1:
        print(f"Move disk 1: {src} -> {dst}")
        return 1
    moves  = hanoi(n-1, src, aux, dst)    # move n-1 to aux
    print(f"Move disk {n}: {src} -> {dst}")
    moves += hanoi(n-1, aux, dst, src)    # move n-1 from aux to dst
    return moves + 1

total = hanoi(3)
print(f"Total moves: {total}")    # 7  (= 2^3 - 1)

%         Permutations of a string


def permutations(s):
    if len(s) <= 1:
        return [s]
    result = []
    for i, ch in enumerate(s):
        rest = s[:i] + s[i+1:]
        for perm in permutations(rest):
            result.append(ch + perm)
    return result

print(sorted(permutations("ABC")))
# ['ABC','ACB','BAC','BCA','CAB','CBA']  (3! = 6)

#         GCD (Euclidean algorithm)
```

```python
def gcd(a, b):
    if b == 0: return a
    return gcd(b, a % b)

print(gcd(48, 18))   # 6
# Call trace: gcd(48,18) -> gcd(18,12) -> gcd(12,6) -> gcd(6,0) = 6

#         Exponentiation by squaring O(log n) calls

@lru_cache(maxsize=None)
def fast_pow(base, exp):
    if exp == 0: return 1
    if exp % 2 == 0:
        half = fast_pow(base, exp // 2)
        return half * half       # only 1 recursive call!
    return base * fast_pow(base, exp - 1)

print(fast_pow(2, 32))   # 4294967296
print(fast_pow.cache_info())

#         GATE question: count total calls for fib(6)

# Answer: calls(n) = calls(n-1) + calls(n-2) + 1
# calls(0)=1, calls(1)=1
# calls(2)=3, calls(3)=5, calls(4)=9, calls(5)=15, calls(6)=25
# fib(6) makes 25 total function calls (without memoisation)

#         GATE question: what is the output?

def mystery(n):
    if n <= 0: return 0
    return mystery(n-2) + n

print(mystery(6))   # mystery(4)+6 = mystery(2)+4+6 = mystery(0)+2+4+6 = 0+12 = 12
print(mystery(5))   # mystery(3)+5 = mystery(1)+3+5 = mystery(-1)+1+3+5 = 0+9 = 9
```

4.6 Lambda Functions

4.6.0.1 Anonymous Single-Expression Functions

Lambda — Syntax and Semantics

A **lambda** is an anonymous, single-expression function created with the `lambda` keyword:

`lambda` ⟨*params*⟩`:` ⟨*expression*⟩

- The body must be a **single expression** (not statements like `if/else` blocks, `for` loops, or `return`)
- The expression's value is **implicitly returned** (no `return` keyword)
- Produces a **function object**, identical in capability to a `def` — same scoping rules, closures, and first-class status
- Can have **default arguments**: `lambda x, n=2: x**n`
- Can have ***args** and ****kwargs**
- Ternary expression *is* allowed (it is an expression): `lambda x: x if x > 0 else -x`

When to use lambda: short, throwaway functions passed as arguments. Prefer `def` when the function is complex, reused, or needs a docstring.

lambda

```
f = lambda x, y:  x + y
# anonymous, one expression
```

≡

def

```
def f(x, y):
    return x + y
# named, can have statements
```

Both produce a **function object**: same type, same LEGB scoping, same closure behaviour, same first-class status

Lambda Properties and Constraints

- **Allowed**: expressions, ternary (`a if c else b`), nested lambdas, `*args`, `**kwargs`, default args, function calls, comprehensions
- **Not allowed**: statements (`print` is a function in Python 3, so `lambda: print("hi")` is fine; but `for`, `while`, `if/elif/else` *blocks*, `return`, `assert`, `raise` are not)
- **Debugging**: lambda functions show as `<lambda>` in tracebacks — hard to debug; prefer `def` in complex code
- **Late binding**: same closure behaviour as nested `def` (loop lambda bug applies equally)

Example 119: Lambda — All Syntactic Forms

```python
# Basic
square = lambda x: x ** 2
print(square(5))            # 25
print(type(square))         # <class 'function'>
print(square.__name__)      # <lambda>

# Multiple parameters
add  = lambda x, y: x + y
dist = lambda x1, y1, x2, y2: ((x2-x1)**2 + (y2-y1)**2)**0.5
print(add(3, 4))            # 7
print(dist(0,0,3,4))        # 5.0

# Default arguments
power = lambda x, n=2: x ** n
print(power(3))             # 9   (n=2 default)
print(power(3, 3))          # 27

# Ternary in lambda body
absolute = lambda x: x if x >= 0 else -x
sign     = lambda x: 1 if x > 0 else (-1 if x < 0 else 0)
print(absolute(-5), sign(-3), sign(0), sign(7))   # 5 -1 0 1

# *args and **kwargs
vsum  = lambda *args: sum(args)
greet = lambda **kw: f"Hello, {kw.get('name','World')}!"
print(vsum(1,2,3,4,5))          # 15
print(greet(name="Alice"))       # Hello, Alice!

# Nested lambda (lambda returning lambda)
make_adder = lambda n: (lambda x: x + n)
add5 = make_adder(5)
print(add5(10))             # 15

# Immediately invoked lambda (IIFE)
print((lambda x, y: x * y)(6, 7))   # 42

# Sorting key (most common use)
students = [("Alice",85),("Bob",72),("Charlie",91),("Diana",85)]
by_score  = sorted(students, key=lambda s: s[1], reverse=True)
by_name   = sorted(students, key=lambda s: s[0])
tiebreak  = sorted(students, key=lambda s: (-s[1], s[0]))
print(by_score)    # [('Charlie',91),('Alice',85),('Diana',85),('Bob',72)]
print(tiebreak)    # [('Charlie',91),('Alice',85),('Diana',85),('Bob',72)]
```

4.7 Higher-Order Functions

4.7.1 map()

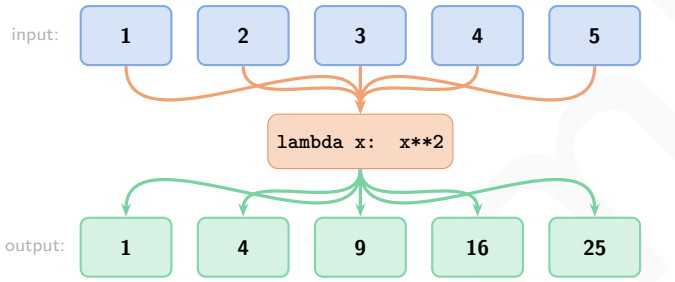
4.7.1.1 Applying a Function to Every Element

map() — Definition

`map(func, iterable, ...)` applies `func` to each element of `iterable` and returns a **lazy iterator** (not a list).

$$\text{map}(f, [x_1, x_2, \ldots, x_n]) \longrightarrow (f(x_1), f(x_2), \ldots, f(x_n))$$

Multiple iterables: `map(f, it1, it2)` passes pairs `(e1, e2)` to `f(e1, e2)`; stops at the shortest iterable.
Lazy: elements are computed *only when consumed*. Use `list(map(...))` to materialise, or iterate with `for`.



Example 120: map() — Single and Multiple Iterables

```
# Basic: single iterable
nums = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, nums))
print(squares)              # [1, 4, 9, 16, 25]

# Using a named function
def celsius_to_f(c): return c * 9/5 + 32
temps_c = [0, 20, 37, 100]
temps_f = list(map(celsius_to_f, temps_c))
print(temps_f)              # [32.0, 68.0, 98.6, 212.0]

# String operations
words = ["hello", "WORLD", "python"]
print(list(map(str.upper,  words)))   # ['HELLO', 'WORLD', 'PYTHON']
print(list(map(str.title,  words)))   # ['Hello', 'World', 'Python']
print(list(map(len, words)))          # [5, 5, 6]

# Multiple iterables: stops at shortest
a = [1, 2, 3, 4]
b = [10, 20, 30]
print(list(map(lambda x,y: x+y, a, b)))   # [11, 22, 33]  (4 ignored)
print(list(map(pow, [2,3,4], [3,2,1])))   # [8, 9, 4]  2^3, 3^2, 4^1

# map is lazy: no computation until consumed
m = map(lambda x: (print(f"  computing {x}"), x**2)[1], range(3))
print("map object created")   # no computation yet
print(next(m))                # computing 0  ->  0
print(next(m))                # computing 1  ->  1
print(list(m))                # computing 2  ->  [4]

# map vs list comprehension (equivalent)
result_map  = list(map(lambda x: x*2, range(5)))
result_comp = [x*2 for x in range(5)]
print(result_map == result_comp)   # True
# List comprehension generally preferred for readability
```

4.7.2 filter()

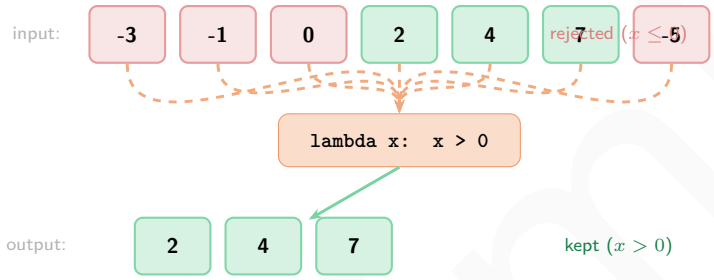
4.7.2.1 Selecting Elements by Predicate

filter() — Definition

`filter(func, iterable)` returns an iterator of elements for which `func(element)` is **truthy**.

$$\text{filter}(p, [x_1, \ldots, x_n]) \longrightarrow (x_i \mid p(x_i) \text{ is truthy})$$

Special case: `func=None` uses the element's own truthiness (removes all falsy values: `0`, `""`, `[]`, `None`, `False`, ...).
Like `map`, `filter` is **lazy**.



Example 121: filter() — Predicates and None

```python
data = [-3, -1, 0, 2, 4, 7, -5]

# Filter with lambda
positives = list(filter(lambda x: x > 0, data))
print(positives)        # [2, 4, 7]

evens = list(filter(lambda x: x % 2 == 0, data))
print(evens)            # [0, 2, 4]

# filter(None, ...) removes falsy values
mixed = [0, 1, "", "hi", [], [1,2], None, False, True, 0.0, 3.14]
print(list(filter(None, mixed)))
# [1, 'hi', [1, 2], True, 3.14]

# Filtering strings
words = ["apple", "banana", "cherry", "avocado", "blueberry"]
a_words = list(filter(lambda w: w.startswith("a"), words))
print(a_words)     # ['apple', 'avocado']

long_words = list(filter(lambda w: len(w) > 6, words))
print(long_words) # ['banana', 'avocado', 'blueberry']

# filter + map: filter then transform
scores = [45, 82, 33, 91, 67, 55, 88]
grade_A = list(map(lambda s: f"{s}->A",
                   filter(lambda s: s >= 80, scores)))
print(grade_A)     # ['82->A', '91->A', '88->A']

# Equivalent comprehension (usually preferred)
grade_A2 = [f"{s}->A" for s in scores if s >= 80]
print(grade_A == grade_A2)  # True
```

4.7.3 functools.reduce()

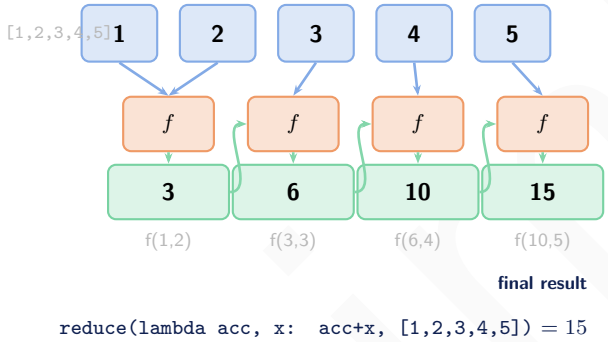
4.7.3.1 Folding an Iterable to a Single Value

reduce() — Definition

`functools.reduce(func, iterable[, initializer])` reduces an iterable to a **single value** by applying `func(accumulator, current)` repeatedly from left to right:

$$\text{reduce}(f, [x_1, x_2, x_3, x_4]) = f(f(f(x_1, x_2), x_3), x_4)$$

Initializer: if provided, it is placed *before* the iterable elements and serves as the starting accumulator value. Required when the iterable might be empty.
Without initializer + empty iterable: raises `TypeError`.



`reduce(lambda acc, x: acc+x, [1,2,3,4,5])` $= 15$

Example 122: reduce() — Product, Max, Flatten, Compose

```
from functools import reduce

data = [1, 2, 3, 4, 5]

# Sum (equivalent to sum())
print(reduce(lambda acc, x: acc + x, data))           # 15
print(reduce(lambda acc, x: acc + x, data, 100))      # 115  (initializer)

# Product (no built-in in Python < 3.8)
print(reduce(lambda acc, x: acc * x, data))           # 120 = 5!

# Max (equivalent to max())
print(reduce(lambda a, b: a if a > b else b, data))  # 5

# String concatenation
words = ["Python", " ", "is", " ", "great"]
print(reduce(lambda a, b: a + b, words))              # Python is great

# Flatten list of lists
nested = [[1,2],[3,4],[5,6]]
flat   = reduce(lambda a, b: a + b, nested)
print(flat)   # [1, 2, 3, 4, 5, 6]

# Function composition: compose(f,g)(x) = f(g(x))
double = lambda x: x * 2
inc    = lambda x: x + 1
square = lambda x: x ** 2

def compose(*funcs):
    return reduce(lambda f, g: lambda x: f(g(x)), funcs)

% compose(square, inc, double): x -> double -> inc -> square
transform = compose(square, inc, double)
print(transform(3))   # double(3)=6, inc(6)=7, square(7)=49

% Empty iterable: needs initializer
```

```
print(reduce(lambda a,b: a+b, [], 0))    # 0
try:
    reduce(lambda a,b: a+b, [])           # TypeError: no initial value!
except TypeError as e:
    print(e)

% map + filter + reduce pipeline
data = range(1, 11)
result = reduce(
    lambda acc, x: acc + x,              # reduce: sum
    filter(lambda x: x % 2 == 0,         # filter: evens
           map(lambda x: x**2, data))    # map: squares
)
print(result)    # 2^2+4^2+6^2+8^2+10^2 = 4+16+36+64+100 = 220
```

4.8 Closures & Nested Functions

4.8.1 What is a Closure?

4.8.1.1 Definition and `__closure__` Cells

Closure — Formal Definition

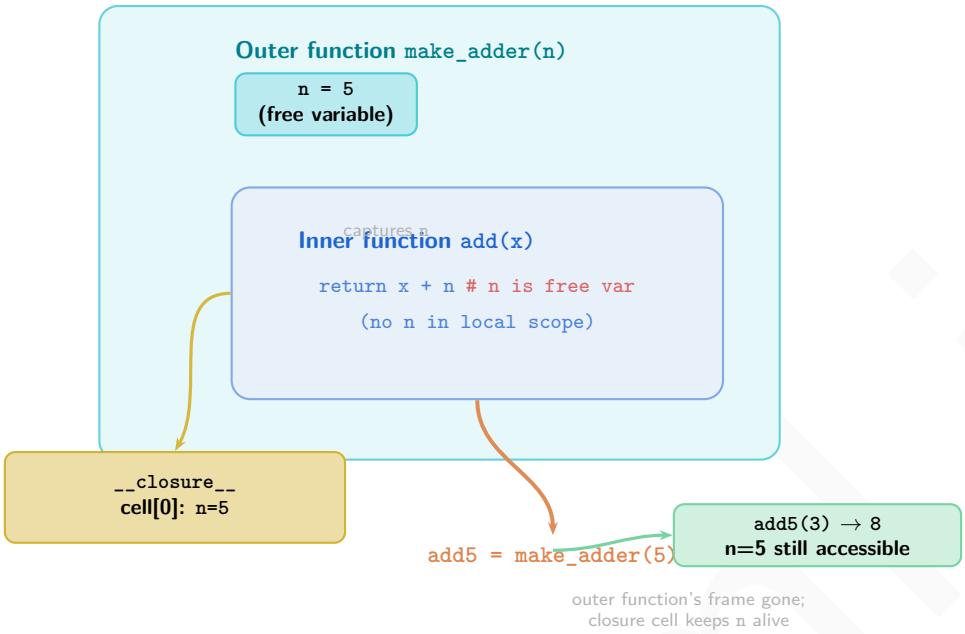
A **closure** is a function object that **remembers values in its enclosing scope** even after program execution has left that scope.

More precisely, a closure is formed when:

1. There is a **nested (inner) function** defined inside an outer function
2. The inner function **references at least one name** from the outer function's local scope (a *free variable*)
3. The outer function **returns** the inner function (or the inner function escapes the outer scope in some other way)

The captured variables are stored as **cell objects** inside the inner function's `__closure__` attribute. Each cell wraps one variable; `cell.cell_contents` retrieves the current value.

Why closures matter: they provide *stateful functions* without requiring a class, and they are the foundation of *decorators*, *factories*, and *memoisation*.



Closure Properties

- `__closure__`: tuple of `cell` objects, or `None` if no free variables
- `cell.cell_contents`: current value stored in that cell
- `func.__code__.co_freevars`: names of the free variables (matches `__closure__` index-for-index)
- A function with no free variables is *not* a closure (`__closure__` is `None`)
- The outer function's **frame is deallocated** after it returns, but the cell objects remain alive as long as the closure exists
- A closure makes the **enclosing variable's lifetime** independent of the enclosing function's call lifetime

Example 123: Closure Basics — Inspecting `__closure__`

```
def make_adder(n):
    """Return a closure that adds n to its argument."""
    def add(x):
        return x + n        # n is a free variable
    return add

add5  = make_adder(5)
add10 = make_adder(10)

print(add5(3))              # 8
print(add10(3))             # 13

# Inspect the closure
print(add5.__closure__)                         # (<cell at 0x...>,)
print(add5.__closure__[0].cell_contents)        # 5
print(add10.__closure__[0].cell_contents)       # 10

print(add5.__code__.co_freevars)                # ('n',)

# A plain function has no closure
def plain(x): return x * 2
print(plain.__closure__)    # None

# Closure keeps outer variable alive
import gc, sys
def outer():
```

```
    big_data = list(range(1000))    # would normally be GC'd after outer() returns
    def inner():
        return len(big_data)        # big_data captured in cell
    return inner

get_len = outer()           # outer() returned; big_data NOT freed
print(get_len())            # 1000   (big_data still alive in closure)
```

Example 124: Closure as Stateful Function (without class)

```
def make_counter(start=0, step=1):
    """Factory: returns a counter closure."""
    count = [start]           # list so mutation works without nonlocal

    def counter():
        val = count[0]
        count[0] += step
        return val

    return counter

c1 = make_counter(0, 1)
c2 = make_counter(10, 5)

print(c1(), c1(), c1())     # 0 1 2
print(c2(), c2())           # 10 15
print(c1())                 # 3  (c1 and c2 are independent!)

# More idiomatic: using nonlocal
def make_counter2(start=0):
    n = start
    def inc():
        nonlocal n
        n += 1
        return n
    def dec():
        nonlocal n
        n -= 1
        return n
    def reset():
        nonlocal n
        n = start
    return inc, dec, reset

inc, dec, rst = make_counter2(0)
print(inc(), inc(), inc())  # 1 2 3
print(dec())                # 2
rst()
print(inc())                # 1
```

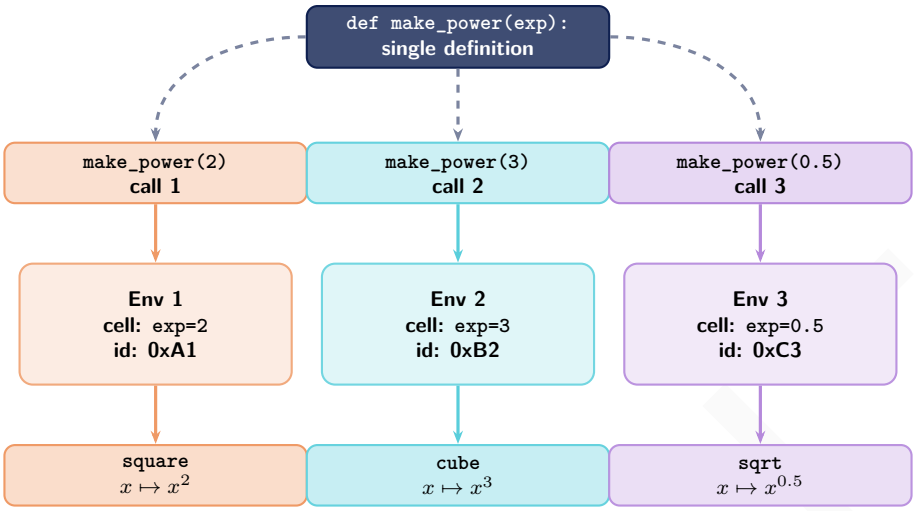
4.8.2 Independent Closure Instances

4.8.2.1 Each Outer Call Creates a New Environment

Independent Closure Instances

Every call to the **outer function** creates a **brand-new, independent set of cell objects**. Two closures returned by two separate calls to the same outer function **never share** their enclosing variables — even though they were created by the same code.

This is the mechanism that makes closures useful as **factories**: the same `def` block produces customised function objects on demand.



Environments 1, 2, 3 are **completely independent**: modifying exp in one does *not* affect the others

Example 125: Independent Closure Instances — Factory Pattern

```
def make_power(exp):
    def power(base):
        return base ** exp
    return power

square = make_power(2)
cube   = make_power(3)
sqrt_  = make_power(0.5)

print(square(4))     # 16
print(cube(4))       # 64
print(sqrt_(4))      # 2.0

# They share NO state
print(square.__closure__[0].cell_contents)   # 2
print(cube.__closure__[0].cell_contents)     # 3  (different cell)
print(square.__closure__ is cube.__closure__)  # False

# Real factory: validators
def make_range_validator(lo, hi):
    def validate(x):
        if lo <= x <= hi:
            return x
        raise ValueError(f"{x} not in [{lo}, {hi}]")
    return validate

check_score  = make_range_validator(0, 100)
check_age    = make_range_validator(0, 150)
check_rating = make_range_validator(1, 5)

print(check_score(85))     # 85
print(check_rating(4))     # 4
try:
    check_score(110)
except ValueError as e:
    print(e)               # 110 not in [0, 100]

# Multiple calls: each closure is truly independent
counters = [make_power(n) for n in range(1, 5)]
for p in counters:
    print(p(2), end=" ")   # 2 4 8 16
```

4.8.3 Shared Mutable State Within One Closure

4.8.3.1 Multiple Inner Functions, One Shared Cell

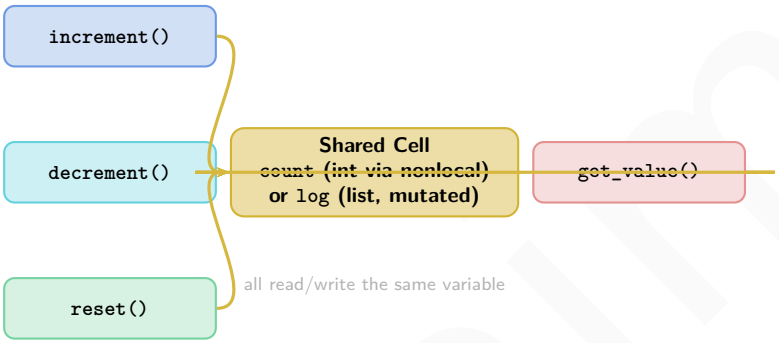
Shared Mutable State in a Closure

When **multiple inner functions** are defined inside the *same* outer function, they share the **same cell objects** for any variables from that outer scope.

If the closed-over variable is a **mutable object** (list, dict), *all* inner functions see every mutation made by any of them. This is the basis of *shared-state closures* — an OOP-like pattern using only functions.

Important distinction:

- Sharing *the same cell* ⇒ rebinding with `nonlocal` is visible to all inner functions
- Sharing *the same mutable object* ⇒ mutations are visible without `nonlocal`



Example 126: Shared Mutable State — Stack and Bank Account

```
#         Shared state via nonlocal (immutable int)

def make_bank_account(initial=0):
    balance = initial

    def deposit(amount):
        nonlocal balance
        if amount <= 0:
            raise ValueError("Deposit must be positive")
        balance += amount
        return balance

    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            raise ValueError("Insufficient funds")
        balance -= amount
        return balance

    def get_balance():
        return balance          # read only: no nonlocal needed

    return deposit, withdraw, get_balance

dep, wdr, bal = make_bank_account(1000)
print(dep(500))     # 1500
print(wdr(200))     # 1300
print(bal())        # 1300  (all three share the same 'balance' cell)

#         Shared state via mutable object (no nonlocal needed)
def make_history_tracker():
    log = []            # mutable: shared without nonlocal

    def record(event):
```

```
        log.append({"event": event, "n": len(log)+1})

    def undo():
        if log:
            return log.pop()
        return None

    def show():
        for entry in log:
            print(f"  [{entry['n']}] {entry['event']}")

    return record, undo, show

rec, undo, show = make_history_tracker()
rec("login")
rec("query_db")
rec("update_profile")
show()
# [1] login
# [2] query_db
# [3] update_profile
print(undo())   # {'event': 'update_profile', 'n': 3}
show()
# [1] login
# [2] query_db

#         Two separate outer calls share NOTHING

dep2, wdr2, bal2 = make_bank_account(0)   # fresh account
dep2(100)
print(bal())    # 1300  (first account: unchanged)
print(bal2())   # 100   (second account: independent)
```

4.8.4 Late Binding in Closures

4.8.4.1 The Loop Closure Bug

Late Binding — Closures Capture Variables, Not Values

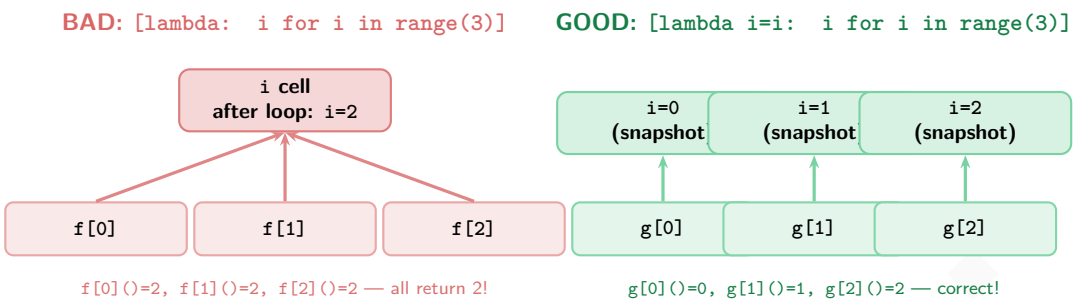
Python closures capture the **variable** (the cell reference) — *not the value* at the time the closure is created. When the closure is *called*, it looks up the *current* value of the variable in the cell.

The loop closure bug is the most common manifestation:

1. A loop creates multiple closures (e.g. via `lambda` or `def` inside a `for`)
2. All closures capture the *same loop variable*
3. After the loop ends, the loop variable holds its *final* value
4. Calling any of the closures returns the *final* value — not the value at iteration time

Fixes:

- **Default argument** `lambda x, i=i: ...` — evaluates `i` at *definition time* and binds it as a default
- **Immediately Invoked Function** (IIFE): `(lambda i=i: lambda x: x*i)()` — same principle
- **Explicit factory function**: `def make(i): return lambda x: x*i`



Example 127: Late Binding Bug — All Three Fixes

```python
#         BUG: late binding captures variable, not value
funcs_bad = [lambda: i for i in range(5)]
print([f() for f in funcs_bad])
# [4, 4, 4, 4, 4]  ALL return 4 (final value of i)

#         FIX 1: Default argument (most Pythonic)
funcs_fix1 = [lambda i=i: i for i in range(5)]
print([f() for f in funcs_fix1])
# [0, 1, 2, 3, 4]  correct!

#         FIX 2: Explicit factory function
def make_func(i):
    return lambda: i         # i is now a local in make_func

funcs_fix2 = [make_func(i) for i in range(5)]
print([f() for f in funcs_fix2])
# [0, 1, 2, 3, 4]  correct!

#         FIX 3: functools.partial


from functools import partial

def get_val(i): return i
funcs_fix3 = [partial(get_val, i) for i in range(5)]
print([f() for f in funcs_fix3])
# [0, 1, 2, 3, 4]  correct!

#         The bug extends to any mutable loop variable
names = ["Alice", "Bob", "Charlie"]
greets_bad  = [lambda: f"Hello, {name}!" for name in names]
greets_good = [lambda name=name: f"Hello, {name}!" for name in names]

print([g() for g in greets_bad])
# ['Hello, Charlie!', 'Hello, Charlie!', 'Hello, Charlie!']

print([g() for g in greets_good])
# ['Hello, Alice!', 'Hello, Bob!', 'Hello, Charlie!']

#         More subtle: variable mutated AFTER closure creation
x = 10
get_x = lambda: x           # captures 'x' from enclosing scope
print(get_x())              # 10
x = 99                      # mutate AFTER closure was created
print(get_x())              # 99  (sees the NEW value! late binding)
```

Example 128: GATE-Level Closure Tracing Questions

```python
# Q1: What does this print?
def outer(x):
    def inner(y):
        return x + y
    return inner

f = outer(10)
g = outer(20)
print(f(5))         # 15  (x=10 from f's closure)
print(g(5))         # 25  (x=20 from g's closure)
print(f(g(1)))      # f(21) = 10+21 = 31

# Q2: Trace with nonlocal and shared cell
def make_pair():
    n = 0
    def inc(): nonlocal n; n += 1; return n
    def dec(): nonlocal n; n -= 1; return n
    return inc, dec

a, b = make_pair()
print(a())   # 1
print(a())   # 2
print(b())   # 1
print(a())   # 2  (n was 1, inc -> 2)

# Q3: Late binding with multiplication
multipliers = [lambda x: x * i for i in range(1, 4)]
print(multipliers[0](5))   # 15  (NOT 5! i is 3 at call time)
print(multipliers[1](5))   # 15
print(multipliers[2](5))   # 15

# Q4: closure cell identity
def outer2():
    val = []
    def f(): val.append(1)
    def g(): val.append(2)
    return f, g, lambda: val

f2, g2, get = outer2()
f2(); g2(); f2()
print(get())   # [1, 2, 1]  all three share same list object

# Q5: What is closure of inner here?
def make(a, b):
    def inner():
        return a * b
    return inner

h = make(3, 4)
print(h.__code__.co_freevars)     # ('a', 'b')
print(h.__closure__[0].cell_contents)  # 3  (a)
print(h.__closure__[1].cell_contents)  # 4  (b)
print(h())   # 12

# Q6: Default arg vs closure --- crucial difference
def create():
    results = []
    for n in [1, 2, 3]:
        # WRONG: lambda captures n by reference
        results.append(lambda: n)
    return results

fns = create()
print([f() for f in fns])   # [3, 3, 3]  (n=3 after loop)

def create_fixed():
    results = []
    for n in [1, 2, 3]:
        results.append(lambda n=n: n)   # snapshot n
```

```
    return results

fns2 = create_fixed()
print([f() for f in fns2])  # [1, 2, 3]
```

4.8.4.2 Closure vs Class — When to Use Which

Closure vs Class — Comparison

Feature	Closure	Class
Single behaviour	Simple	Possible
Multiple behaviours	Possible	Natural
Readable when small	Very	Verbose
Inheritance	No	Yes
Pickle/serialise	Limited	Full
Introspect state	Hard	Easy (`self.x`)
No boilerplate	Yes	No

Rule of thumb: use a **closure** for a single callable with private state; use a **class** when you need multiple methods, inheritance, serialisation, or clear state introspection.

Example 129: Same Problem: Closure vs Class

```
#         Closure approach


def make_moving_average():
    data = []
    def update(x):
        data.append(x)
        return sum(data) / len(data)
    return update

avg = make_moving_average()
print(avg(10))     # 10.0
print(avg(20))     # 15.0
print(avg(30))     # 20.0

#         Class approach (equivalent)


class MovingAverage:
    def __init__(self):
        self._data = []

    def update(self, x):
        self._data.append(x)
        return sum(self._data) / len(self._data)

    @property
    def history(self):            # extra: easy introspection
        return list(self._data)

```

```
avg2 = MovingAverage()
print(avg2.update(10))    # 10.0
print(avg2.update(20))    # 15.0
print(avg2.history)       # [10, 20]   -- not possible with closure easily
```

4.9 Decorators

4.9.1 Concept

4.9.1.1 What a Decorator Is

Decorator — Definition

A **decorator** is a **callable that takes a function (or class) and returns a new callable**. It *wraps* the original function to add behaviour before or after (or instead of) the original call.

Syntactic sugar:

```
@decorator          ≡    def f():  ...
def f():  ...            f = decorator(f)
```

The `@` syntax is applied at **function definition time**, not at call time.

```
@syntax
@my_decorator
def greet():
   print("Hello")
```

≡

```
explicit form
def greet():
   print("Hello")
greet = my_decorator(greet)
```

Definition time: `my_decorator(greet)` is called → returns wrapper → name greet now points to wrapper

4.9.2 Writing a Decorator

4.9.2.1 Wrapper Pattern and functools.wraps

Standard Decorator Pattern

```
def my_decorator(func):  # 1.  receives original function

    def wrapper(*args, **kwargs):  # 2.  defines the wrapper

        # 3.  add behaviour BEFORE
        result = func(*args, **kwargs) # 4.  call original
        # 5.  add behaviour AFTER
        return result

    return wrapper # 6.  return wrapper (replaces func)
```

functools.wraps — Why It Matters

Without `@wraps`, the wrapped function **loses its identity**: `__name__` becomes `"wrapper"`, `__doc__` is lost. `@functools.wraps(func)` copies `__name__`, `__doc__`, `__module__`, `__qualname__`, `__annotations__`, `__dict__` from the wrapped function to the wrapper.

Always use `@functools.wraps(func)` in production decorators.

Example 130: Writing Decorators — Timer, Logger, Retry

```python
import time, functools

#         1. Timer decorator


def timer(func):
    @functools.wraps(func)           # preserves __name__, __doc__
    def wrapper(*args, **kwargs):
        start  = time.perf_counter()
        result = func(*args, **kwargs)
        end    = time.perf_counter()
        print(f"[{func.__name__}] took {end-start:.4f}s")
        return result
    return wrapper

@timer
def slow_sum(n):
    """Sum 0..n-1."""
    return sum(range(n))

print(slow_sum(10_000_000))    # [slow_sum] took 0.3xxx s
                               # 49999995000000

print(slow_sum.__name__)       # slow_sum   (not 'wrapper'!)
print(slow_sum.__doc__)        # Sum 0..n-1.

#         2. Logger decorator


def logger(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}({args}, {kwargs})")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result!r}")
        return result
    return wrapper

@logger
def add(x, y): return x + y

add(3, y=4)
# Calling add((3,), {'y': 4})
# add returned 7

#         3. Retry decorator


def retry(max_attempts=3):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Attempt {attempt} failed: {e}")
                    if attempt == max_attempts:
                        raise
```

```
        return wrapper
    return decorator

# (decorator with arguments -- see next subsection)
```

4.9.3 Stacking Decorators

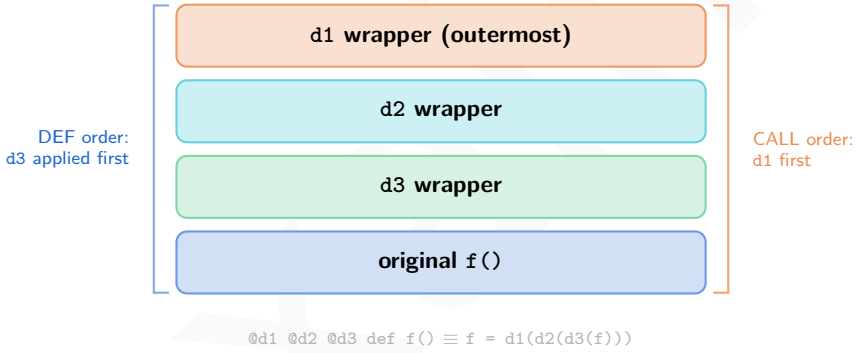
4.9.3.1 Bottom-Up Application Order

Stacking Decorators — Application and Call Order

Multiple decorators stack **bottom-up at definition time**:

```
@d1
@d2          ≡   f = d1(d2(d3(f)))
@d3
def f():  ...
```

Definition: d3 applied first, then d2, then d1.
Call: d1's wrapper runs first, then d2's, then d3's, then the original `f`.



Example 131: Stacking Decorators — Order Proof

```
import functools

def decorator_factory(name):
    """Creates a decorator that announces itself."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print(f"[{name}] BEFORE")
            result = func(*args, **kwargs)
            print(f"[{name}] AFTER")
            return result
        return wrapper
    return decorator

d1 = decorator_factory("d1")
d2 = decorator_factory("d2")
d3 = decorator_factory("d3")

@d1
@d2
@d3
def greet(name):
    print(f"  Hello, {name}!")

greet("Alice")
```

```
# [d1] BEFORE
# [d2] BEFORE
# [d3] BEFORE
#   Hello, Alice!
# [d3] AFTER
# [d2] AFTER
# [d1] AFTER

# Definition is: greet = d1(d2(d3(greet)))
# d3 applied first (innermost), d1 outermost
# At call: d1's wrapper runs first, d2's next, d3's next, then original
```

4.9.4 Decorator with Arguments

4.9.4.1 Three-Level Nesting: Factory → Decorator → Wrapper

Parameterised Decorator — Three Levels

To pass arguments to a decorator, add one more level of nesting: a **decorator factory** that accepts the arguments and returns a decorator:

$$\underbrace{\text{repeat}(3)}_{\text{factory call}} \longrightarrow \underbrace{\text{decorator}}_{\text{receives func}} \longrightarrow \underbrace{\text{wrapper}}_{\text{wraps call}}$$

`@repeat(3) def f()` $\equiv$ `f = repeat(3)(f)`

```
def repeat(n):  # level 1:  factory - takes decorator args
    def decorator(func):  # level 2:  decorator - takes func
        def wrapper(*args, **kwargs):
            for _ in range(n):  func(*args, **kwargs) # level 3:  wrapper
        return wrapper
    return decorator
```

Example 132: Decorators with Arguments — repeat, validate_types, rate_limit

```
import functools, time

#         1. repeat(n): call function n times

def repeat(n):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            results = [func(*args, **kwargs) for _ in range(n)]
            return results[-1]    # return last result
        return wrapper
    return decorator

@repeat(3)
def hello(name):
    print(f"Hello, {name}!")
```

```python

hello("Bob")
# Hello, Bob!
# Hello, Bob!
# Hello, Bob!

#         2. Type validation decorator

def validate_types(**expected):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(**kwargs):
            for key, typ in expected.items():
                if key in kwargs and not isinstance(kwargs[key], typ):
                    raise TypeError(
                        f"Arg '{key}' expected {typ.__name__}, "
                        f"got {type(kwargs[key]).__name__}")
            return func(**kwargs)
        return wrapper
    return decorator

@validate_types(name=str, age=int)
def register(name, age):
    print(f"Registered: {name}, {age}")

register(name="Alice", age=25)    # OK
try:
    register(name="Bob", age="25")  # TypeError!
except TypeError as e:
    print(e)   # Arg 'age' expected int, got str

#         3. rate_limit(calls_per_sec)

def rate_limit(calls_per_sec):
    interval = 1.0 / calls_per_sec
    def decorator(func):
        last_called = [0.0]         # mutable cell trick
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            elapsed = time.time() - last_called[0]
            if elapsed < interval:
                time.sleep(interval - elapsed)
            last_called[0] = time.time()
            return func(*args, **kwargs)
        return wrapper
    return decorator

@rate_limit(calls_per_sec=2)   # max 2 calls per second
def api_call(endpoint):
    return f"response from {endpoint}"

#         4. Combining @wraps with argument inspection

import inspect

def log_calls(level="INFO"):
    def decorator(func):
        sig = inspect.signature(func)
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            bound = sig.bind(*args, **kwargs)
            bound.apply_defaults()
            print(f"[{level}] {func.__name__}({dict(bound.arguments)})")
            result = func(*args, **kwargs)
            print(f"[{level}] returned: {result!r}")
            return result
        return wrapper
    return decorator

@log_calls(level="DEBUG")
```

```
def divide(a, b=1):
    return a / b

divide(10, b=2)
# [DEBUG] divide({'a': 10, 'b': 2})
# [DEBUG] returned: 5.0
```

4.9.5 Common Built-in Decorators

4.9.5.1 functools.lru_cache, cached_property

Built-in Decorators Reference

Decorator	Module	Purpose
`@staticmethod`	built-in	Method with no `self`/`cls`
`@classmethod`	built-in	Method receives class as first arg
`@property`	built-in	Computed attribute (getter/setter/deleter)
`@lru_cache(maxsize=N)`	`functools`	Cache return values (LRU eviction)
`@cache`	`functools`	Unbounded cache (Python 3.9+)
`@cached_property`	`functools`	Computes once, cached on instance dict
`@total_ordering`	`functools`	Fill in comparison methods from `__eq__` + one
`@dataclass`	`dataclasses`	Auto-generate `__init__`, `__repr__`, etc.
`@abstractmethod`	`abc`	Mark method as abstract (must override)

Example 133: cached_property and total_ordering

```
from functools import cached_property, total_ordering
import math

#         cached_property: computed once, stored on instance
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @cached_property
    def area(self):
        print("  computing area...")
        return math.pi * self.radius ** 2

    @cached_property
    def circumference(self):
        print("  computing circumference...")
        return 2 * math.pi * self.radius

c = Circle(5)
print(c.area)               # computing area... / 78.539...
print(c.area)               # (cached: no "computing" print)
print("area" in c.__dict__) # True -- stored in instance dict

#         total_ordering: define __eq__ + one comparison
@total_ordering
class Student:
    def __init__(self, name, gpa):
```

```
        self.name = name
        self.gpa  = gpa

    def __eq__(self, other):
        return self.gpa == other.gpa

    def __lt__(self, other):
        return self.gpa < other.gpa

    # @total_ordering auto-generates: __le__, __gt__, __ge__

s1 = Student("Alice", 3.8)
s2 = Student("Bob",   3.5)
s3 = Student("Charlie", 3.8)

print(s1 > s2)    # True   (auto-generated __gt__)
print(s1 >= s3)   # True   (auto-generated __ge__)
print(s2 <= s1)   # True   (auto-generated __le__)
var_students = [s1, s2, s3]
print(sorted(var_students, key=lambda s: s.gpa))
# Bob(3.5), Alice(3.8), Charlie(3.8)
```

Example 134: Decorator Patterns — GATE Summary Questions

```
# Q1: What does this print?
def d(f):
    def w():
        print("A")
        f()
        print("B")
    return w

@d
def hello(name):
    print("C")

hello()
# A / C / B

# Q2: Stacking order
def p(c):
    def dec(f):
        def w(*a,**k):
            print(c, end=" ")
            return f(*a,**k)
        return w
    return dec

@p("1") @p("2") @p("3")
def fn(): print("X")
fn()   # 1 2 3 X  (d1 runs first, prints 1; d2 prints 2; d3 prints 3)

# Q3: wraps vs no wraps
def bad(f):
    def wrapper(): return f()
    return wrapper

def good(f):
    @functools.wraps(f)
    def wrapper(): return f()
    return wrapper

@bad
def foo(): """foo docstring"""; pass
@good
def bar(): """bar docstring"""; pass

print(foo.__name__)   # wrapper  (LOST original name)
```

```
print(bar.__name__)   # bar      (preserved by @wraps)
print(foo.__doc__)    # None
print(bar.__doc__)    # bar docstring

# Q4: decorator = factory call at definition time
import functools

def count_calls(func):
    @functools.wraps(func)
    def wrapper(*a, **k):
        wrapper.calls += 1
        return func(*a, **k)
    wrapper.calls = 0    # attach counter to wrapper function!
    return wrapper

@count_calls
def add(x, y): return x + y

add(1,2); add(3,4); add(5,6)
print(add.calls)   # 3

\newpage
```

4.10 Problems

Problem 175 ***[MCQ]*** *What is the output?*

```
def f(x):
    return x * 2

print(f(3), f(3) + 1)
```

(A) `6 6`

(B) `6 7`

(C) `3 4`

(D) `6 13`

Problem 176 ***[MCQ]*** *What is the output?*

```
def f():
    x = 10

result = f()
print(result)
```

(A) `10`

(B) `0`

(C) `None`

(D) `NameError`

Problem 177 ***[MCQ]*** *What is the output?*

```
def multi():
    return 1, 2, 3

a = multi()
print(type(a), a)
```

(A) `<class 'list'> [1, 2, 3]`

(B) `<class 'tuple'> (1, 2, 3)`

(C) `<class 'int'> 1`

(D) `<class 'tuple'> 1, 2, 3`

Problem 178 ***[MCQ]*** *What is the output?*

```
def greet(name):
    """Returns a greeting."""
    return "Hi " + name

print(greet.__doc__)
```

(A) `None`

(B) `Returns a greeting.`

(C) `Hi`

(D) `AttributeError`

Problem 179 ***[MCQ]*** *What is the output?*

```
def square(n):
    return n ** 2

f = square
print(f(5))
```

(A) *NameError*

(B) *25*

(C) *10*

(D) *TypeError*

Problem 180 ***[MCQ]*** *What is the output?*

```
def apply(func, val):
    return func(val)

print(apply(abs, -7))
```

(A) *-7*

(B) *7*

(C) *abs*

(D) *TypeError*

Problem 181 ***[MCQ]*** *What is the output?*

```
def f(a, b, c):
    print(a, b, c)

f(1, c=3, b=2)
```

(A) *1 2 3*

(B) *1 3 2*

(C) *TypeError*

(D) *SyntaxError*

Problem 182 ***[MCQ]*** *What is the output?*

```
def f(x, y=5, z=10):
    return x + y + z

print(f(1, z=2))
```

(A) *13*

(B) *8*

(C) *18*

(D) *TypeError*

Problem 183 ***[MCQ]*** *What is the output?*

```
def f(*args):
    return type(args), len(args)

print(f(1, 2, 3))
```

(A) *(<class 'list'>, 3)*

(B) *(<class 'tuple'>, 3)*

(C) *(<class 'tuple'>, 1)*

(D) *TypeError*

Problem 184 ***[MCQ]*** *What is the output?*

```
def f(**kwargs):
    return kwargs

print(f(a=1, b=2))
```

(A) (1, 2)

(B) ['a', 'b']

(C) {'a': 1, 'b': 2}

(D) TypeError

Problem 185 ***[MCQ]*** *What is the output?*

```
def f(a, *args, b):
    print(a, args, b)

f(1, 2, 3, b=4)
```

(A) 1 (2, 3) 4

(B) 1 2 3 4

(C) 1 (2, 3, 4)

(D) TypeError: b is missing

Problem 186 ***[MCQ]*** *What is the output?*

```
def f(a, *, b):
    return a + b

print(f(3, b=4))
print(f(3, 4))
```

(A) 7 then 7

(B) 7 then TypeError

(C) TypeError for both

(D) 7 then SyntaxError

Problem 187 ***[MCQ]*** *What is the output?*

```
def f(a, b, /, c):
    return a + b + c

print(f(1, 2, c=3))
print(f(1, b=2, c=3))
```

(A) 6 then 6

(B) 6 then TypeError

(C) TypeError for both

(D) 6 then SyntaxError

Problem 188 ***[MCQ]*** *What is the output?*

```
def f(x, y, z):
    print(x, y, z)

args = (1, 2, 3)
f(*args)
```

(A) (1, 2, 3)

(B) `1 2 3`

(C) `TypeError`

(D) `[1, 2, 3]`

Problem 189 ***[MCQ]*** *What is the output?* ***(Classic GATE trap)***

```
def f(x, lst=[]):
    lst.append(x)
    return lst

print(f(1))
print(f(2))
print(f(3))
```

(A) `[1]  [2]  [3]`

(B) `[1]  [1, 2]  [1, 2, 3]`

(C) `[1, 2, 3]  [1, 2, 3]  [1, 2, 3]`

(D) `TypeError`

Problem 190 ***[MCQ]*** *What is the output?*

```
def f(x, lst=[]):
    lst.append(x)
    return lst

a = f(10)
b = f(20)
print(a is b)
```

(A) `False`, *because* `a` *and* `b` *hold different lists*

(B) `True`, *because both calls share the same default list object*

(C) `False`, *because integers are immutable*

(D) `TypeError`

Problem 191 ***[MCQ]*** *Which of the following fixes the mutable default argument problem?*

```
def f(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst
```

(A) *It does not fix the problem; the bug persists.*

(B) *It fixes the problem: a new list is created on every call that uses the default.*

(C) *It fixes the problem, but only for the first call.*

(D) `None` *is also mutable, so the same problem applies.*

Problem 192 ***[MCQ]*** *What is the output?*

```
def f(x, lst=[]):
    lst.append(x)
    return lst

print(f(1))
print(f(2, []))
print(f(3))
```

(A) `[1]  [2]  [1, 3]`

(B) `[1]  [2]  [3]`

(C) `[1]  [1, 2]  [1, 3]`

(D) `[1]  [2]  [1, 2, 3]`

Problem 193 ***[MSQ]*** *Which of the following default argument types are* ***unsafe*** *(shared across calls) in Python?*

(A) `def f(x=[])`

(B) `def f(x={})`

(C) `def f(x=0)`

(D) `def f(x=set())`

(E) `def f(x=None)`

Problem 194 ***[MCQ]*** *What is the output?*

```
data = [1, 4, 9, 16, 25]
if (n := len(data)) > 3:
    print(f"List has {n} elements")
```

(A) `List has 3 elements`

(B) `List has 5 elements`

(C) *No output*

(D) `SyntaxError`

Problem 195 ***[MCQ]*** *What is the primary benefit of the walrus operator :=?*

(A) *It provides a faster alternative to == for comparison.*

(B) *It assigns a value* ***and*** *returns it in a single expression, avoiding recomputation.*

(C) *It creates a new local variable scope.*

(D) *It is used for augmented assignment like +=.*

Problem 196 ***[MCQ]*** *What is the output?*

```
def f(lst):
    lst.append(99)

L = [1, 2, 3]
f(L)
print(L)
```

(A) `[1, 2, 3]`

(B) `[1, 2, 3, 99]`

(C) `[99]`

(D) `None`

Problem 197 ***[MCQ]*** *What is the output?*

```
def f(lst):
    lst = [10, 20, 30]

L = [1, 2, 3]
f(L)
print(L)
```

(A) `[10, 20, 30]`

(B) `[1, 2, 3, 10, 20, 30]`

(C) `[1, 2, 3]`

(D) []

Problem 198 ***[MCQ]*** *What is the output?*

```
def f(x):
    x += 10
    return x

a = 5
b = f(a)
print(a, b)
```

(A) 5 5

(B) 15 15

(C) 5 15

(D) 15 5

Problem 199 ***[MCQ]*** *What is the output?*

```
def f(d):
    d["key"] = "new"

data = {"key": "old"}
f(data)
print(data)
```

(A) {'key': 'old'}

(B) {'key': 'new'}

(C) {}

(D) TypeError

Problem 200 ***[MCQ]*** *What is the output?*

```
x = 10

def f():
    print(x)

f()
```

(A) None

(B) 10

(C) NameError

(D) UnboundLocalError

Problem 201 ***[MCQ]*** *What is the output?* ***(Classic GATE trap)***

```
x = 10

def f():
    print(x)
    x = 20

f()
```

(A) 10

(B) 20

(C) None

(D) UnboundLocalError

Problem 202 ***[MCQ]*** *What is the output?*

```
x = 1

def f():
    global x
    x = 2

f()
print(x)
```

(A) `1`

(B) `2`

(C) `None`

(D) `NameError`

Problem 203 ***[MCQ]*** *What is the output?*

```
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 99
    inner()
    print(x)

outer()
```

(A) `10`

(B) `99`

(C) `None`

(D) `NameError`

Problem 204 ***[MCQ]*** *What is the output?*

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        print(x)
    inner()

outer()
```

(A) `global`

(B) `enclosing`

(C) `None`

(D) `NameError`

Problem 205 ***[MCQ]*** *What is the output?*

```
def outer():
    count = 0
    def inner():
        count += 1
        return count
    return inner()

print(outer())
```

(A) `1`

(B) `0`

(C) `None`

(D) `UnboundLocalError`

Problem 206 ***[MCQ]*** *What is the output?*

```
def make_adder(n):
    def adder(x):
        return x + n
    return adder

add5 = make_adder(5)
add10 = make_adder(10)
print(add5(3), add10(3))
```

(A) `8 13`

(B) `8 8`

(C) `13 13`

(D) `NameError`

Problem 207 ***[MCQ]*** *What is the output?* ***(Late-binding trap — very frequently tested)***

```
funcs = []
for i in range(3):
    funcs.append(lambda: i)

print(funcs[0](), funcs[1](), funcs[2]())
```

(A) `0 1 2`

(B) `2 2 2`

(C) `0 0 0`

(D) `1 1 1`

Problem 208 ***[MCQ]*** *What is the output? (Late-binding fix using default argument)*

```
funcs = []
for i in range(3):
    funcs.append(lambda i=i: i)

print(funcs[0](), funcs[1](), funcs[2]())
```

(A) `2 2 2`

(B) `0 1 2`

(C) `0 0 0`

(D) `TypeError`

Problem 209 ***[MCQ]*** *What is the output?*

```
def counter():
    count = [0]
    def inc():
        count[0] += 1
        return count[0]
    return inc

c1 = counter()
c2 = counter()
print(c1(), c1(), c2())
```

(A) 1 2 1

(B) 1 1 1

(C) 1 2 3

(D) UnboundLocalError

Problem 210 ***[MCQ]*** *What is the output?*

```
def outer():
    data = []
    def push(x):
        data.append(x)
    def peek():
        return data[-1] if data else None
    return push, peek

push, peek = outer()
push(10)
push(20)
print(peek())
```

(A) 10

(B) 20

(C) None

(D) IndexError

Problem 211 ***[MCQ]*** *Why does the following code print 2 2 2 instead of 0 1 2?*

```
funcs = [lambda: i for i in range(3)]
print(funcs[0](), funcs[1](), funcs[2]())
```

(A) List comprehensions do not support lambdas.

(B) All lambdas capture the same variable `i`, which equals 2 after the loop ends.

(C) The lambda body is evaluated at definition, not at call time.

(D) `range(3)` generates 2 for all values.

Problem 212 ***[NAT]***

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)

print(fact(5))
```

How many times is `fact` called (including the initial call) when `fact(4)` is evaluated using the function above?

Problem 213 ***[MCQ]*** *What is the output?*

```
def f(n):
    if n <= 0:
        return 0
    return f(n - 1) + f(n - 2) + 1

print(f(4))
```

(A) 4

(B) 6

(C) 7

(D) 8

Problem 214 ***[MCQ]*** *Consider the following recursive function. What is g(5)?*

```
def g(n):
    if n == 1:
        return 1
    if n % 2 == 0:
        return g(n // 2)
    return g(n - 1) + 1
```

(A) 2

(B) 3

(C) 4

(D) 5

Problem 215 ***[MCQ]*** *What is the output?*

```
def mystery(s):
    if len(s) <= 1:
        return s
    return mystery(s[1:]) + s[0]

print(mystery("gate"))
```

(A) gate

(B) etag

(C) eagte

(D) etga

Problem 216 ***[MCQ]*** *What is the output?* ***(GATE DA 2024 style)***

```
def fun(D, s1, s2):
    if s1 < s2:
        D[s1], D[s2] = D[s2], D[s1]
        fun(D, s1 + 1, s2 - 1)

D = [1, 2, 3, 4, 5]
fun(D, 0, 4)
print(D)
```

(A) [1, 2, 3, 4, 5]

(B) [5, 4, 3, 2, 1]

(C) [5, 2, 3, 4, 1]

(D) [1, 4, 3, 2, 5]

Problem 217 ***[MCQ]*** *What is the output?*

```
def h(n, acc=0):
    if n == 0:
        return acc
    return h(n - 1, acc + n)

print(h(5))
```

(A) 5

(B) 15

(C) 10

(D) 0

Problem 218 ***[MCQ]*** *What does the following recursive function compute for a positive integer n?*

```
def f(n):
    if n == 0:
        return 0
    return f(n // 10) + n % 10
```

(A) The number of digits in n

(B) The sum of digits of n

(C) The reverse of n

(D) The largest digit of n

Problem 219 ***[NAT]*** *The sum of total number of cache hits and cache misses is*

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(5))
```

Problem 220 ***[MCQ]*** *What is the output?*

```
square = lambda x: x ** 2
print(square(7))
```

(A) 14

(B) 49

(C) 7

(D) TypeError

Problem 221 ***[MCQ]*** *What is the output?*

```
data = [(1, 'b'), (3, 'a'), (2, 'c')]
data.sort(key=lambda t: t[1])
print(data)
```

(A) [(1, 'b'), (2, 'c'), (3, 'a')]

(B) [(3, 'a'), (1, 'b'), (2, 'c')]

(C) [(1, 'b'), (3, 'a'), (2, 'c')]

(D) [(3, 'a'), (2, 'c'), (1, 'b')]

Problem 222 ***[MCQ]*** *What is the output?*

```
f = lambda x, n=3: x ** n
print(f(2), f(2, 2))
```

(A) 6 4

(B) 8 4

(C) 8 8

(D) TypeError

Problem 223 ***[MCQ]*** *What is the output?*

```
result = list(map(lambda x: x ** 2, [1, 2, 3, 4]))
print(result)
```

(A) `[1, 4, 9, 16]`

(B) `[2, 4, 6, 8]`

(C) `[1, 2, 3, 4]`

(D) `map object`

Problem 224 ***[MCQ]*** *What is the output?*

```
result = list(filter(lambda x: x % 2 == 0, range(10)))
print(result)
```

(A) `[1, 3, 5, 7, 9]`

(B) `[0, 2, 4, 6, 8]`

(C) `[2, 4, 6, 8]`

(D) `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

Problem 225 ***[NAT]*** *What is the output?*

```
from functools import reduce
result = reduce(lambda acc, x: acc * x, [1, 2, 3, 4, 5])
print(result)
```

Problem 226 ***[MCQ]*** *What is the output?*

```
from functools import reduce
result = reduce(lambda a, b: a + b, [1, 2, 3, 4], 10)
print(result)
```

(A) `10`

(B) `20`

(C) `21`

(D) `ValueError`

Problem 227 ***[MCQ]*** *What is the output?*

```
result = list(filter(None, [0, 1, "", "hi", [], [1], None, True]))
print(result)
```

(A) `[0, "", [], None]`

(B) `[1, "hi", [1], True]`

(C) `[1, "hi", True]`

(D) `[True]`

Problem 228 ***[MCQ]*** *What is the output?*

```
a = [1, 2, 3]
b = [10, 20, 30]
result = list(map(lambda x, y: x + y, a, b))
print(result)
```

(A) `[1, 2, 3, 10, 20, 30]`

(B) `[11, 22, 33]`

(C) `[(1, 10), (2, 20), (3, 30)]`

(D) `TypeError`

Problem 229 ***[MCQ]*** *What is the output?*

```
def deco(func):
    def wrapper(*args, **kwargs):
        print("before")
        result = func(*args, **kwargs)
        print("after")
        return result
    return wrapper

@deco
def greet():
    print("hello")

greet()
```

(A) `hello`

(B) `before hello after`

(C) `before after`

(D) `TypeError`

Problem 230 ***[MCQ]*** *What is the output?*

```
def deco(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) * 2
    return wrapper

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

print(add(3, 4))
```

(A) `7`

(B) `14`

(C) `TypeError`

(D) `28`

Problem 231 ***[MCQ]*** *What is the output?* ***(Stacked decorators)***

```
def d1(f):
    def w():
        print("d1 in", end=" ")
        f()
        print("d1 out", end=" ")
    return w

def d2(f):
    def w():
        print("d2 in", end=" ")
        f()
        print("d2 out", end=" ")
    return w

@d1
@d2
def hello():
    print("hi", end=" ")

hello()
```

(A) `d2 in hi d2 out d1 in d1 out`

(B) `d1 in d2 in hi d2 out d1 out`

(C) `d1 in hi d1 out`

(D) `d2 in hi d2 out`

Problem 232 ***[MCQ]*** *The expression `@d1 @d2 def f(): ...` is equivalent to which of the following?*

(A) `f = d2(d1(f))`

(B) `f = d1(d2(f))`

(C) `f = d1(f); f = d2(f)`

(D) *None*

Problem 233 ***[MCQ]*** *What is the output?*

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def say():
    print("hi", end=" ")

say()
```

(A) `hi`

(B) `hi hi hi`

(C) `TypeError`

(D) *No output*

Problem 234 ***[MCQ]*** *What is the output?*

```
def f(n):
    return (lambda x: x * n)(3)

print(f(4))
```

(A) `12`

(B) `7`

(C) `3`

(D) `TypeError`

Problem 235 ***[NAT]*** *What is the output of the following code?*

```
def f(a, b, *args, **kwargs):
    return len(args) + len(kwargs)

print(f(1, 2, 3, 4, x=5, y=6))
```

Problem 236 ***[MCQ]*** *What is the output?*

```
ops = {
    "add": lambda a, b: a + b,
    "mul": lambda a, b: a * b,
}
print(ops["add"](3, 4), ops["mul"](3, 4))
```

(A) `7 12`

(B) `12 7`

(C) `3 4`

(D) `TypeError`

Problem 237 ***[MSQ]*** *Which of the following are* ***correct*** *about Python closures?*

(A) A closure captures free variables by reference, not by value.

(B) Each call to the outer function creates an independent closure environment.

(C) The captured variables are stored in the function's `__closure__` *attribute.*

(D) A closure can modify an enclosing variable without `nonlocal` *if the variable is a mutable object (e.g. list).*

(E) Two closures from the same outer-function call always share the same closed-over variables.

4.11 YouTube Links and QR Codes

Lecture	Details	YouTube Link	QR Code
15	Basics of Functions in Python	https://youtu.be/WQ2vF8XtCH4	
16	Scope & LEGB Rule in Python	https://youtu.be/NfdbmkHJGOg	
17	Recursion in Python	https://youtu.be/L0c5hz7kcvQ	
18	Memoization in Recursion	https://youtu.be/iifzFtYT1SI	
19	Lambda & Higher Order Functions	https://youtu.be/OTKpUZT8g6c	

20	Closures in Python	https://youtu.be/XRUhpxeX5v8	
21	Decorators in Python	https://youtu.be/J_ClO6BAdUc	
22	Problem Solving on Functions	https://youtu.be/xVWfcSaqhBE	

Chapter 5

Comprehensions & Generators

5.1 List Comprehensions

5.1.1 Syntax, Multiple for Clauses, Nesting

List Comprehension — Syntax and Semantics

A **list comprehension** builds a new list by applying an expression to each element of an iterable, optionally filtering with a condition:

$$\texttt{[}\ \underbrace{\textit{expr}}_{\text{output}}\ \texttt{for}\ \underbrace{x}_{\text{variable}}\ \texttt{in}\ \underbrace{\textit{iterable}}_{\text{source}}\ \underbrace{\texttt{if}\ \textit{condition}}_{\text{optional filter}}\ \texttt{]}$$

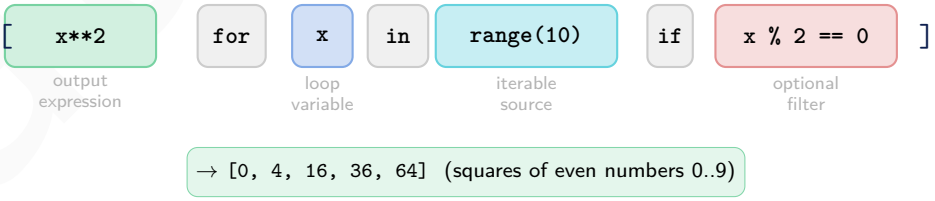
Equivalent `for` loop:

```
result = []  for x in iterable:    if condition:   result.append(expr)
```

Multiple `for` clauses: evaluated left-to-right like nested `for` loops:

```
[f(x,y) for x in A for y in B]  ≡  for x in A: for y in B: append(f(x,y))
```

Produces a new list, evaluated **eagerly** (immediately).



List Comprehension Patterns

- **Simple transform**: `[f(x) for x in it]`
- **Filtered transform**: `[f(x) for x in it if p(x)]`
- **Conditional value**: `[a if cond else b for x in it]` (ternary in expression position, not filter)
- **Cartesian product**: `[(x,y) for x in A for y in B]` — outer loop first

- **Flatten 2D**: `[x for row in matrix for x in row]`
- **Nested list comp**: `[[f(x) for x in row] for row in matrix]` — creates 2D result; outer loop is the outer `for`
- **Walrus in filter**: `[y for x in it if (y:=f(x)) > 0]` — compute once, use in both filter and output

Example 135: List Comprehensions — All Patterns

```python
# 1. Basic transform
squares = [x**2 for x in range(1, 6)]
print(squares)       # [1, 4, 9, 16, 25]

# 2. Filter
evens = [x for x in range(20) if x % 2 == 0]
print(evens)         # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# 3. Conditional expression (ternary in output, NOT a filter)
clipped = [x if x > 0 else 0 for x in [-3,-1,0,2,5,-4,8]]
print(clipped)       # [0, 0, 0, 2, 5, 0, 8]

# 4. Cartesian product: both for clauses
pairs = [(x,y) for x in [1,2,3] for y in ['a','b']]
print(pairs)
# [(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]

# 5. Flatten 2D matrix
matrix = [[1,2,3],[4,5,6],[7,8,9]]
flat   = [x for row in matrix for x in row]
print(flat)          # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 6. Multiple conditions
nums = [x for x in range(100) if x % 3 == 0 if x % 5 == 0]
print(nums)          # [0, 15, 30, 45, 60, 75, 90]

# 7. Nested list comprehension (2D result)
transposed = [[row[i] for row in matrix] for i in range(3)]
print(transposed)    # [[1,4,7],[2,5,8],[3,6,9]]

# 8. Walrus operator: avoid double computation
import math
results = [root for x in range(20) if (root := math.sqrt(x)) > 3.5]
print(results[:4])   # [3.872..., 4.0, 4.123..., ...]

# 9. String processing
words  = ["hello", "world", "python", "gate", "exam"]
titled = [w.title() for w in words if len(w) > 4]
print(titled)        # ['Hello', 'World', 'Python']

# 10. Dict/set from comprehension intermediary
matrix2 = [[1,0,3],[0,5,0],[7,0,9]]
non_zero = [(i,j) for i,row in enumerate(matrix2)
                  for j,v  in enumerate(row) if v != 0]
print(non_zero)     # [(0,0),(0,2),(1,1),(2,0),(2,2)]
```

5.2 Dictionary Comprehensions

5.2.1 Key-Value Mapping Expressions

Dictionary Comprehension — Syntax

$$\{\ \underbrace{k:v}_{\text{key:value}}\ \texttt{for}\ \langle vars\rangle\ \texttt{in}\ \langle iterable\rangle\ [\texttt{if}\ \langle condition\rangle]\ \}$$

Produces a **new dict**; if the same key appears more than once, the *last* value wins (later iteration overwrites earlier).

Example 136: Dictionary Comprehensions — Transform, Filter, Invert

```python
# 1. Transform values
nums = {"a": 1, "b": 2, "c": 3}
squared = {k: v**2 for k, v in nums.items()}
print(squared)        # {'a': 1, 'b': 4, 'c': 9}

# 2. Filter entries
passing = {k: v for k, v in nums.items() if v > 1}
print(passing)        # {'b': 2, 'c': 3}

# 3. Invert: value -> key (assumes unique values)
orig = {"a": 1, "b": 2, "c": 3}
inv  = {v: k for k, v in orig.items()}
print(inv)            # {1: 'a', 2: 'b', 3: 'c'}

# 4. From two lists (zip)
keys   = ["name", "age", "city"]
values = ["Alice",  25,  "Delhi"]
d = {k: v for k, v in zip(keys, values)}
print(d)    # {'name': 'Alice', 'age': 25, 'city': 'Delhi'}

# 5. Character frequency
text = "abracadabra"
freq = {ch: text.count(ch) for ch in set(text)}
print(freq)    # {'r':2,'c':1,'a':5,'d':1,'b':2}

# 6. Conditional key transformation
words = ["hello", "WORLD", "Python", "GATE"]
normalised = {w.lower(): len(w) for w in words}
print(normalised)  # {'hello':5,'world':5,'python':6,'gate':4}

# 7. Nested dict comprehension
matrix = [[1,2,3],[4,5,6],[7,8,9]]
indexed = {(i,j): matrix[i][j]
           for i in range(3) for j in range(3)}
print(indexed[(1,2)])   # 6

# 8. Duplicate key: last wins
data = [("a",1),("b",2),("a",3),("b",4)]
d2 = {k: v for k, v in data}
print(d2)    # {'a': 3, 'b': 4}  (last 'a' and 'b' values win)
```

5.3 Set Comprehensions

5.3.1 Unique-Element Expressions

Set Comprehension — Syntax

{ ⟨*expr*⟩ `for` ⟨*var*⟩ `in` ⟨*iterable*⟩ [`if` ⟨*condition*⟩] }

Automatically deduplicates: equivalent to `set([... comprehension ...])`. **Unordered**: iteration order is not guaranteed. **Note**: `{expr}` without `for` is a *set literal*; empty set must be `set()`, not `{}`.

Example 137: Set Comprehensions

```python
# 1. Unique squares
sq = {x**2 for x in range(-4, 5)}
print(sorted(sq))     # [0, 1, 4, 9, 16]  (deduped: (-2)^2 == 2^2)

# 2. Unique first letters
words = ["apple","ant","banana","blueberry","avocado"]
initials = {w[0] for w in words}
print(initials)       # {'a', 'b'}

# 3. Common elements (set comprehension on two lists)
a = [1,2,3,4,5,3,2]
b = [3,4,5,6,7,4,3]
common = {x for x in a if x in b}
print(common)         # {3, 4, 5}

# 4. Characters appearing in both strings
s1, s2 = "abcde", "cdefg"
shared = {c for c in s1 if c in s2}
print(shared)         # {'c', 'd', 'e'}

# 5. Deduplication with transformation
data = [3, -3, 1, -1, 2, -2, 1]
unique_abs = {abs(x) for x in data}
print(unique_abs)     # {1, 2, 3}
```

5.4 Generator Expressions

5.4.1 Lazy Iterators from Expressions

Generator Expression — Syntax and Laziness

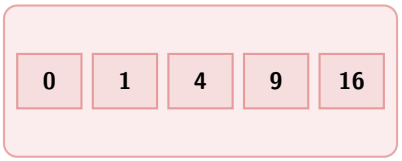
A **generator expression** has the same syntax as a list comprehension but uses **parentheses** (or no delimiters when the sole argument to a function):

(⟨*expr*⟩ `for` x `in` ⟨*iterable*⟩ [`if` ⟨*cond*⟩])

Key differences from list comprehension:

	List comp `[...]`	**Generator expr** `(...)`
Evaluation	Eager (immediate)	Lazy (on demand)
Result type	`list`	`generator` iterator
Memory	All elements at once	One element at a time
Reusable	Yes	**No** (exhausted after one pass)
Indexable	Yes (`res[0]`)	No

List comp `[x**2 for x in range(5)]`



5 integers in memory immediately

Generator expr `(x**2 for x in range(5))`

only next value computed when `next()` called

`sys.getsizeof([x**2 for x in range(1000)])` ≈ **8056 B** **vs** `sys.getsizeof((x**2 for x in range(1000)))` ≈ **104 B**

Example 138: Generator Expressions — Memory, Pipeline, Exhaustion

```python
import sys

# Memory: generator vs list comprehension
lst = [x**2 for x in range(1000)]
gen = (x**2 for x in range(1000))
print(sys.getsizeof(lst))   # ~8056 bytes
print(sys.getsizeof(gen))   # ~104 bytes (just the generator object)

# Generator as argument (no extra parens needed)
total = sum(x**2 for x in range(1001))    # sum of squares 0..1000
print(total)    # 333833500

any_neg = any(x < 0 for x in [1, 2, -3, 4])
print(any_neg)  # True  (short-circuits at -3!)

all_pos = all(x > 0 for x in [1, 2, 3, 4])
print(all_pos)  # True

# Lazy pipeline: large file simulation
def read_lines():
    for i in range(1, 6):
        yield f"line {i}: data={i*10}"

lines   = (line for line in read_lines())         # lazy source
numbers = (int(line.split("=")[1]) for line in lines)  # lazy parse
big     = (n for n in numbers if n > 20)                # lazy filter
result  = list(big)                                     # materialise
print(result)   # [30, 40, 50]

# Exhaustion: generator can only be iterated ONCE
g = (x for x in range(3))
print(list(g))   # [0, 1, 2]
print(list(g))   # []  EMPTY -- generator exhausted!

# Compare with list (reusable)
lst2 = [x for x in range(3)]
print(list(lst2))   # [0, 1, 2]
print(list(lst2))   # [0, 1, 2]  still works!

```

```python
# next() on generator
g2 = (x**2 for x in range(5))
print(next(g2))    # 0
print(next(g2))    # 1
print(next(g2))    # 4
print(list(g2))    # [9, 16]  rest consumed
```

5.5 Generator Functions

5.5.1 yield Statement

5.5.2 Suspending and Resuming Execution

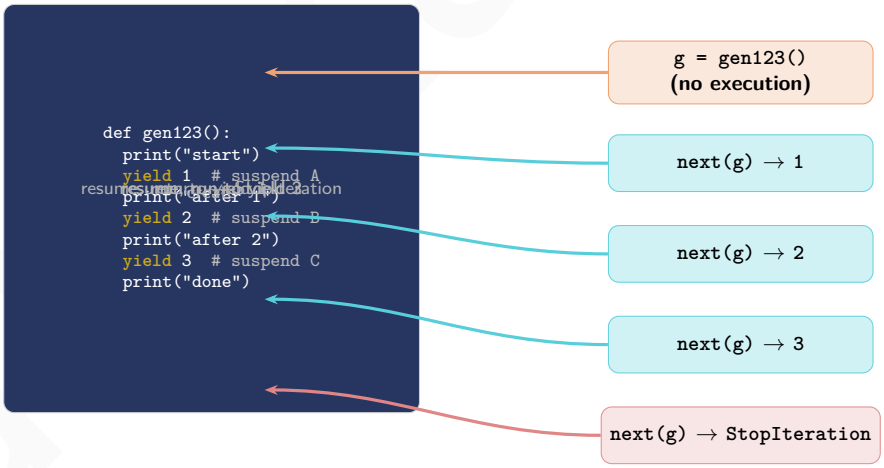
yield — Suspending a Function

Any function containing a `yield` statement becomes a **generator function**. Calling it returns a **generator object** (an iterator) without executing any of the function body.

Execution model:

1. Calling `gen_func()` returns a generator object; body not run yet
2. `next(gen)` runs until the next `yield`, suspends, and returns the yielded value
3. `next(gen)` again resumes from after the `yield`
4. When the function **returns** (or falls off the end), a `StopIteration` is raised; `return value` sets `StopIteration.value`

State preserved between calls: local variables, loop counters, execution pointer — the frame is kept alive, not torn down.



Example 139: Generator Functions — Basics and Infinite Streams

```python
# Basic generator
def gen123():
    print("start")
    yield 1
    print("after 1")
    yield 2
    print("after 2")
    yield 3
    print("done")

```

```python
g = gen123()
print(type(g))        # <class 'generator'>
print(next(g))        # start / 1
print(next(g))        # after 1 / 2
print(next(g))        # after 2 / 3
try:
    next(g)           # done / StopIteration
except StopIteration:
    print("exhausted")

# for loop consumes automatically
for v in gen123():
    print(v)          # start / 1 / after 1 / 2 / after 2 / 3 / done

# return in generator: sets StopIteration.value
def gen_with_return():
    yield 1
    yield 2
    return "final"    # StopIteration.value = "final"

g2 = gen_with_return()
print(next(g2))       # 1
print(next(g2))       # 2
try:
    next(g2)
except StopIteration as e:
    print(e.value)    # final

# Infinite generator (only possible with generator; not with list!)
def natural_numbers(start=1):
    n = start
    while True:
        yield n
        n += 1

gen = natural_numbers()
print([next(gen) for _ in range(7)])    # [1, 2, 3, 4, 5, 6, 7]

# Fibonacci generator
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
print([next(fib) for _ in range(10)])
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

# Practical: large file reading without loading all into memory
def read_chunks(text, size=5):
    for i in range(0, len(text), size):
        yield text[i:i+size]

for chunk in read_chunks("Hello World Python GATE", size=5):
    print(repr(chunk), end=" ")
# 'Hello' ' Worl' 'd Pyt' 'hon G' 'ATE'
```

5.6 Problems

Problem 238 ***[MCQ]*** *What is the output?*

```
result = [x ** 2 for x in range(5)]
print(result)
```

(A) `[1, 4, 9, 16, 25]`

(B) `[0, 1, 4, 9, 16]`

(C) `[0, 1, 2, 3, 4]`

(D) `[1, 2, 3, 4, 5]`

Problem 239 ***[MCQ]*** *What is the output?*

```
result = [x for x in range(10) if x % 3 == 0]
print(result)
```

(A) `[3, 6, 9]`

(B) `[0, 3, 6, 9]`

(C) `[1, 4, 7]`

(D) `[0, 3, 6]`

Problem 240 ***[MCQ]*** *What is the output?*

```
result = [x if x % 2 == 0 else -x for x in range(6)]
print(result)
```

(A) `[0, 2, 4]`

(B) `[0, -1, 2, -3, 4, -5]`

(C) `[-1, -3, -5]`

(D) `[0, 1, 2, 3, 4, 5]`

Problem 241 ***[MCQ]*** *What is the output?*

```
A = [1, 2]
B = [3, 4]
result = [x + y for x in A for y in B]
print(result)
```

(A) `[4, 5, 5, 6]`

(B) `[4, 6]`

(C) `[4, 5, 6]`

(D) `[(1,3),(1,4),(2,3),(2,4)]`

Problem 242 ***[MCQ]*** *What is the output?*

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
```

(A) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

(B) `[1, 2, 3, 4, 5, 6, 7, 8, 9]`

(C) `[1, 4, 7, 2, 5, 8, 3, 6, 9]`

(D) `[7, 8, 9]`

Problem 243 ***[NAT]*** *What is the length of the list produced by the following comprehension?*

```
result = [x * y for x in range(1, 4) for y in range(1, 4) if x != y]
```

Problem 244 ***[MCQ]*** *What is the output?*

```
words = ["hello", "world", "python"]
result = [w.upper() for w in words if len(w) > 5]
print(result)
```

(A) ['HELLO', 'WORLD', 'PYTHON']

(B) ['HELLO', 'WORLD']

(C) ['PYTHON']

(D) ['HELLO', 'WORLD', 'PYTHON']

Problem 245 ***[MCQ]*** *What is the output?*

```
result = [[i * j for j in range(1, 4)] for i in range(1, 4)]
print(result[1])
```

(A) [1, 2, 3]

(B) [2, 4, 6]

(C) [3, 6, 9]

(D) [1, 4, 9]

Problem 246 ***[MCQ]*** *What is the output?*

```
data = [1, -2, 3, -4, 5]
result = [abs(x) for x in data if x < 0]
print(result)
```

(A) [1, 3, 5]

(B) [-2, -4]

(C) [2, 4]

(D) [1, 2, 3, 4, 5]

Problem 247 ***[MCQ]*** *What is the output?*

```
d = {x: x ** 2 for x in range(1, 6)}
print(d)
```

(A) {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

(B) {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

(C) [1, 4, 9, 16, 25]

(D) {1, 4, 9, 16, 25}

Problem 248 ***[MCQ]*** *What is the output?*

```
d = {"a": 1, "b": 2, "c": 3}
inv = {v: k for k, v in d.items()}
print(inv)
```

(A) {"a": 1, "b": 2, "c": 3}

(B) {1: "a", 2: "b", 3: "c"}

(C) {("a",1), ("b",2), ("c",3)}

(D) *TypeError*

Problem 249 ***[MCQ]*** *What is the output?*

```
keys = ["x", "y", "x", "z"]
d = {k: i for i, k in enumerate(keys)}
print(d)
```

(A) *{'x': 0, 'y': 1, 'z': 3}*

(B) *{'x': 2, 'y': 1, 'z': 3}*

(C) *{'x': 0, 'y': 1, 'x': 2, 'z': 3}*

(D) *ValueError*

Problem 250 ***[MCQ]*** *What is the output?*

```
s = {x % 4 for x in range(10)}
print(len(s))
```

(A) *10*

(B) *4*

(C) *3*

(D) *5*

Problem 251 ***[MCQ]*** *What is the output?*

```
data = [("a", 1), ("b", 2), ("a", 3)]
d = {k: v for k, v in data}
print(d)
```

(A) *{'a': 1, 'b': 2}*

(B) *{'a': [1, 3], 'b': [2]}*

(C) *{'a': 3, 'b': 2}*

(D) *ValueError: duplicate key*

Problem 252 ***[MCQ]*** *What does the following expression produce?*

```
result = {x for x in "mississippi"}
print(len(result))
```

(A) *11*

(B) *4*

(C) *5*

(D) *3*

Problem 253 ***[MCQ]*** *What is the output?*

```
gen = (x ** 2 for x in range(5))
print(type(gen))
```

(A) *<class 'list'>*

(B) *<class 'tuple'>*

(C) *<class 'generator'>*

(D) *<class 'iterator'>*

Problem 254 ***[MCQ]*** *What is the output?*

```
gen = (x for x in range(5))
print(sum(gen))
print(sum(gen))
```

(A) *10 then 10*

(B) *10 then 0*

(C) *10 then StopIteration*

(D) *0 then 0*

Problem 255 ***[MCQ]*** *What is the output?*

```
total = sum(x * x for x in range(1, 5))
print(total)
```

(A) *16*

(B) *30*

(C) *25*

(D) *14*

Problem 256 ***[MCQ]*** *What is the key difference between `[x**2 for x in range(1000)]` and `(x**2 for x in range(1000))`?*

(A) *The list comprehension returns a tuple; the generator returns a list.*

(B) *The list comprehension is evaluated eagerly and stores all values; the generator is lazy and produces values one at a time.*

(C) *There is no difference in memory usage.*

(D) *The generator expression raises `TypeError` when iterated.*

Problem 257 ***[MCQ]*** *What is the output?*

```
gen = (x for x in range(10) if x % 2 == 0)
print(next(gen))
print(next(gen))
```

(A) *0 then 0*

(B) *0 then 2*

(C) *2 then 4*

(D) *0 then 1*

Problem 258 ***[MCQ]*** *What is the output?*

```
data = [1, 2, 3, 4, 5]
gen = (x for x in data if x > 3)
data.append(6)
print(list(gen))
```

(A) *`[4, 5]`*

(B) *`[4, 5, 6]`*

(C) *`[1, 2, 3, 4, 5, 6]`*

(D) *`[4, 5, 6]` only if evaluated after `append`*

Problem 259 ***[MCQ]*** *What is the output?*

```python
def gen():
    yield 1
    yield 2
    yield 3

g = gen()
print(next(g), next(g))
```

(A) `1 1`

(B) `1 2`

(C) `2 3`

(D) `1 2 3`

Problem 260 ***[MCQ]*** *What is the output?*

```python
def gen():
    print("start")
    yield 1
    print("middle")
    yield 2
    print("end")

g = gen()
next(g)
next(g)
```

(A) `start middle end`

(B) `start` then `middle`

(C) `start` (only)

(D) No output

Problem 261 ***[MCQ]*** *What is the output?*

```python
def gen():
    yield 10
    yield 20

g = gen()
print(list(g))
print(list(g))
```

(A) `[10, 20]` then `[10, 20]`

(B) `[10, 20]` then `[]`

(C) `[10]` then `[20]`

(D) `[10, 20]` then `StopIteration`

Problem 262 ***[MCQ]*** *What happens when `next()` is called on an exhausted generator?*

(A) It returns `None`.

(B) It raises `StopIteration`.

(C) It resets and starts from the beginning.

(D) It returns 0.

Problem 263 ***[MCQ]*** *What is the output?*

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1

g = countdown(4)
print(sum(g))
```

(A) 4

(B) 10

(C) 6

(D) 3

Problem 264 ***[MCQ]*** *What is the output?*

```python
def gen():
    for i in range(3):
        yield i * 2

print(list(gen()))
```

(A) [0, 1, 2]

(B) [0, 2, 4]

(C) [2, 4, 6]

(D) [1, 2, 3]

Problem 265 ***[MCQ]*** *What is the output?*

```python
def gen():
    yield 1
    return
    yield 2

print(list(gen()))
```

(A) [1, 2]

(B) [1]

(C) []

(D) SyntaxError

5.7 YouTube Links and QR Codes

Lecture	Details	YouTube Link	QR Code
23	Comprehensions & Generators	https://youtu.be/_yTnD1lZviM	
24	Problem Solving on Comprehensions & Generators	https://youtu.be/gdQv1SrJgq4	

Chapter 6

Object-Oriented Programming

6.1 Classes & Objects

6.1.1 Class Definition

6.1.2 What is a Class?

Class — What It Is

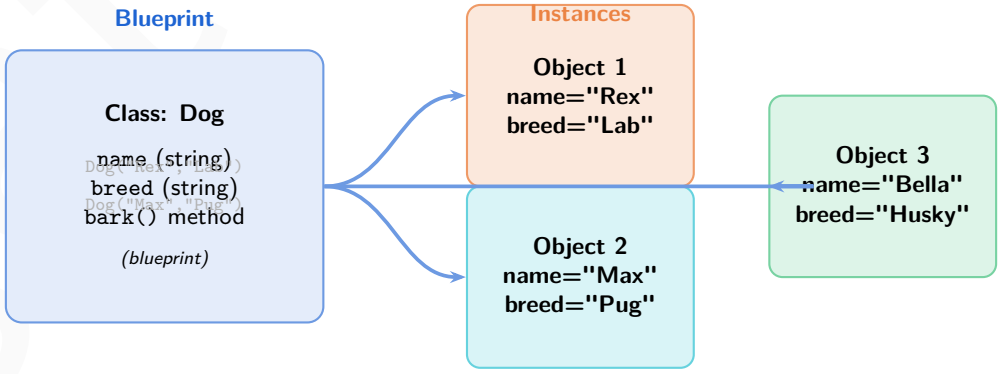
A **class** is like a **blueprint or template**. Just like a house blueprint tells you what every house built from it will have (rooms, doors, windows), a class tells Python what every *object* made from it will have.

An **object** (also called an **instance**) is one real thing built from that blueprint. You can build many objects from one class, just like you can build many houses from one blueprint.

Syntax:

```
class ClassName(Base):
# body:  attributes and methods
```

The class body is run **once** at definition time. A class object is created and stored under the name `ClassName`.



Example 140: Your First Class

```
# Define a class (blueprint)
class Dog:
```

```
    # This runs when we create a Dog object
    def __init__(self, name, breed):
        self.name  = name    # store name on this object
        self.breed = breed   # store breed on this object

    # A method (a function inside a class)
    def bark(self):
        return f"{self.name} says: Woof!"

    def info(self):
        return f"{self.name} is a {self.breed}"

# Create objects (instances) from the class
dog1 = Dog("Rex",   "Labrador")
dog2 = Dog("Max",   "Pug")
dog3 = Dog("Bella", "Husky")

# Use the objects
print(dog1.bark())     # Rex says: Woof!
print(dog2.info())     # Max is a Pug
print(dog3.name)       # Bella

# Each object is independent
dog1.name = "Rocky"    # only dog1 changes
print(dog1.name)       # Rocky
print(dog2.name)       # Max   (unchanged)

# Check type
print(type(dog1))                # <class '__main__.Dog'>
print(isinstance(dog1, Dog))     # True
```

6.1.3 __init__ and self

6.1.3.1 Setting Up an Object

__init__ and self — Simple Explanation

- `__init__` is the **setup method**. Python calls it automatically right after a new object is created. Think of it as the "fill in the details" step.
- `self` is the name for "**this exact object**". When you write `self.name = name`, you are saving the value *on this object*.
- `self` is always the **first parameter** of every instance method. Python passes the object automatically — you do not pass it yourself.
- `__new__` actually *creates* the object (allocates memory); `__init__` just fills in the values. For most purposes you only need `__init__`.

Example 141: __init__ — Setting Up Objects

```
class Student:
    def __init__(self, name, roll, marks):
        # 'self' means 'this student object'
        self.name  = name     # save name on this student
        self.roll  = roll     # save roll number
        self.marks = marks    # save marks

    def grade(self):
        if self.marks >= 90:
            return "A"
        elif self.marks >= 75:
            return "B"
        else:
            return "C"

    def introduce(self):
        return (f"I am {self.name}, "
                f"Roll #{self.roll}, "
                f"Grade: {self.grade()}")

# Create students -- __init__ runs automatically
s1 = Student("Alice", 101, 92)
s2 = Student("Bob",   102, 78)
s3 = Student("Carol", 103, 65)

print(s1.introduce())   # I am Alice, Roll #101, Grade: A
print(s2.introduce())   # I am Bob, Roll #102, Grade: B
print(s3.grade())       # C

# self.marks is different for each student
print(s1.marks, s2.marks, s3.marks)   # 92 78 65

# __init__ is called again for each new object
s4 = Student("Dave", 104, 85)
print(s4.grade())   # B
```

6.1.4 Class Variables vs Instance Variables

6.1.4.1 Shared vs Per-Object Data

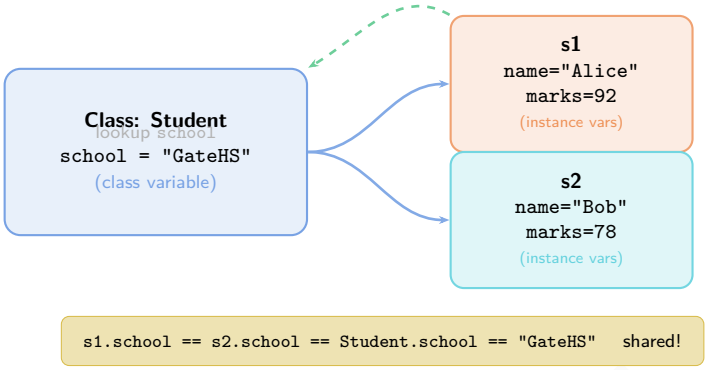
Class Variable vs Instance Variable

- **Class variable**: written *inside the class but outside any method*. **Shared by all objects** of that class. Like a notice board that every student in a class can see.
- **Instance variable**: written using `self.attr` inside a method. **Belongs to one specific object**. Like each student's personal notebook.

Lookup order: Python first looks in the object's own storage (`__dict__`); if not found, it looks in the class's storage.

Important traps:

- **Assignment through instance**: `obj.class_var = x` creates a *new instance variable* that **shadows** the class variable — it does NOT change the class variable.
- **Mutation through instance**: if the class variable is a *mutable* object (list, dict), doing `obj.class_var.append(x)` changes the shared object — all instances see the change.



Example 142: Class vs Instance Variables — The Key Differences

```python
class Student:
    school  = "GateHS"    # class variable: shared by ALL students
    count   = 0           # class variable: how many students created
    history = []          # class variable: MUTABLE -- careful!

    def __init__(self, name, marks):
        Student.count += 1          # update shared counter
        self.name  = name           # instance variable: per student
        self.marks = marks          # instance variable: per student

s1 = Student("Alice", 92)
s2 = Student("Bob",   78)

# Both share the same class variable
print(s1.school)          # GateHS
print(s2.school)          # GateHS
print(Student.school)     # GateHS  (access via class name)
print(Student.count)      # 2  (both students counted)

# Instance variables are different for each
print(s1.name, s1.marks)  # Alice 92
print(s2.name, s2.marks)  # Bob   78

#         TRAP 1: assigning via instance creates a new instance variable
s1.school = "NewSchool"   # creates s1's OWN 'school' (shadows class var!)
print(s1.school)          # NewSchool   (s1's own copy)
print(s2.school)          # GateHS      (class var unchanged)
print(Student.school)     # GateHS      (class var unchanged)
print("school" in s1.__dict__)  # True  (now s1 has its own 'school')

#         TRAP 2: mutating a mutable class variable affects ALL instances
s1.history.append("joined")    # mutates the SHARED list!
print(s1.history)   # ['joined']
print(s2.history)   # ['joined']  s2 also sees it -- same list!
print(Student.history)  # ['joined']
```

6.2 Dunder (Magic) Methods

6.2.0.1 What are Dunder Methods?

Dunder Methods — Simple Explanation

Dunder = **D**ouble **Under**score. Methods like `__init__`, `__str__`, `__add__` have double underscores on both sides.
Python calls these methods **automatically** when you use built-in operations. You define them to teach Python *how your object behaves.*

For example:

- When you write `print(obj)`, Python calls `obj.__str__()`
- When you write `a + b`, Python calls `a.__add__(b)`
- When you write `len(obj)`, Python calls `obj.__len__()`
- When you write `obj[key]`, Python calls `obj.__getitem__(key)`

6.2.1 String Representation: ___str___ and ___repr___

___str___ vs ___repr___

Method	When called	Purpose
`__str__`	`print(obj)`, `str(obj)`, f-strings	Nice, readable text for users
`__repr__`	REPL (just type the name), `repr(obj)`, fallback for `__str__`	Exact description for developers

Simple rule: if you only define one, define `__repr__` — Python uses it as a fallback for `__str__` too.

Example 143: ___str___ and ___repr___

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

    def __str__(self):
        # For humans: nice and clean
        return f"Point({self.x}, {self.y})"

    def __repr__(self):
        # For developers: shows how to recreate it
        return f"Point(x={self.x}, y={self.y})"

p = Point(3, 4)

# __str__ used by print and f-strings
print(p)               # Point(3, 4)
print(f"Location: {p}")   # Location: Point(3, 4)
print(str(p))          # Point(3, 4)

# __repr__ used in REPL and repr()
print(repr(p))         # Point(x=3, y=4)

# Without __str__, Python falls back to __repr__
class Bare:
    def __repr__(self):
        return "I am Bare"

b = Bare()
print(b)               # I am Bare  (__repr__ used as fallback)
print(repr(b))         # I am Bare

# In a list, __repr__ is used for each item
pts = [Point(1,2), Point(3,4)]
print(pts)
# [Point(x=1, y=2), Point(x=3, y=4)]  -- uses __repr__ inside list
```

6.3 Inheritance

6.3.1 Single Inheritance

6.3.1.1 Child Getting Everything from Parent

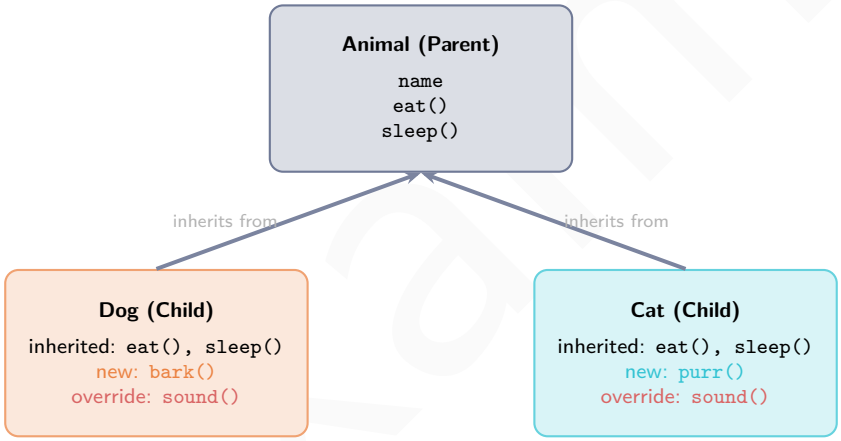
Inheritance — What It Means

Inheritance means a **child class** gets all the methods and attributes of a **parent class** for free. The child can also:

- **Add** new methods and attributes
- **Override** (replace) a parent method with its own version
- **Call the parent's version** using `super()`

Think of it like this: a `Dog` is an `Animal`. Every animal can eat and sleep. A dog can also bark. Instead of writing "eat" and "sleep" again for Dog, we *inherit* them from Animal.

6.3.2 Hierarchical Inheritance



Example 144: Inheritance and super()

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

    def eat(self):
        return f"{self.name} is eating"

    def sleep(self):
        return f"{self.name} is sleeping"

    def sound(self):
        return f"{self.name} makes a sound"

class Dog(Animal):    # Dog inherits from Animal
    def __init__(self, name, breed):
        super().__init__(name)    # call Animal's __init__
        self.breed = breed        # add extra attribute

    def bark(self):               # new method (only for Dog)
        return f"{self.name} says: Woof!"

    def sound(self):              # OVERRIDE: replace Animal's version
        return f"{self.name} says: Woof!"

```

```python
class Cat(Animal):
    def __init__(self, name, indoor):
        super().__init__(name)
        self.indoor = indoor

    def sound(self):              # OVERRIDE: Cat's own version
        return f"{self.name} says: Meow!"

dog = Dog("Rex", "Lab")
cat = Cat("Whiskers", True)

# Inherited methods work
print(dog.eat())      # Rex is eating
print(cat.sleep())    # Whiskers is sleeping

# Overridden method: each uses its own version
print(dog.sound())    # Rex says: Woof!
print(cat.sound())    # Whiskers says: Meow!

# New method only in Dog
print(dog.bark())     # Rex says: Woof!
# cat.bark()          # AttributeError: Cat has no bark()

# isinstance check
print(isinstance(dog, Dog))      # True
print(isinstance(dog, Animal))   # True (Dog IS an Animal)
print(isinstance(cat, Dog))      # False
```

Example 145: super() — Calling Parent from Child

```python
class Shape:
    def __init__(self, colour):
        self.colour = colour

    def describe(self):
        return f"I am a {self.colour} shape"

class Circle(Shape):
    def __init__(self, colour, radius):
        super().__init__(colour)    # runs Shape's __init__
        self.radius = radius

    def describe(self):
        # Call parent's describe, then add more info
        base = super().describe()  # "I am a red shape"
        return f"{base} (circle, radius={self.radius})"

    def area(self):
        return 3.14159 * self.radius ** 2

class Square(Shape):
    def __init__(self, colour, side):
        super().__init__(colour)
        self.side = side

    def area(self):
        return self.side ** 2

c = Circle("red", 5)
s = Square("blue", 4)

print(c.describe())    # I am a red shape (circle, radius=5)
print(s.describe())    # I am a blue shape  (from parent)
print(c.area())        # 78.53975
print(s.area())        # 16
```

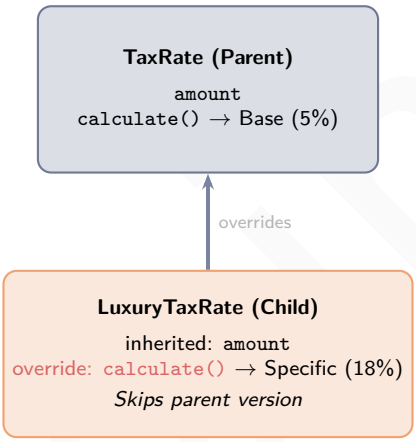
6.3.3 Method Overriding

Method Overriding — Core Concepts

Method overriding occurs when a **child class** provides a specialized implementation for a method that is already defined in its **parent class**. This allows subclasses to customize behavior while maintaining a consistent interface.

- **Exact Match:** The child method must use the exact same name and signature as the parent method.
- **Dynamic Routing:** When invoked on a child object, Python bypasses the parent implementation and executes the child version immediately.
- **Extension via super():** Overriding does not mean abandoning parent logic. You can augment the parent's code by calling `super().method()` inside the child's version.

Think of it like tax systems or reporting frameworks: every report can "generate," but an encrypted report will call the default generation mechanism first and then wrap the result in security tags.



Example 146: Method Overriding and Behavioral Extension

```
class TaxRate:
    def calculate(self, amount):
        return amount * 0.05  # Base default rate (5%)

class LuxuryTaxRate(TaxRate):
    def calculate(self, amount):
        return amount * 0.18  # OVERRIDE: Replace parent rate entirely

# ------------------------------------------------------------
# Framework Extension using super()
# ------------------------------------------------------------
class Report:
    def generate(self):
        return "Base Report Content"

class EncryptedReport(Report):
    def generate(self):
        # 1. Fetch data from base parent framework using super()
        raw_data = super().generate()
        # 2. Add specific child modification layer
        return f"[ENCRYPTED] {raw_data}"

# --- Testing Replacement Overriding ---
standard = TaxRate()
luxury = LuxuryTaxRate()

print(standard.calculate(1000))  # Output: 50.0
print(luxury.calculate(1000))    # Output: 180.0 (Parent calculation is skipped)

```

```python
# --- Testing Extended Overriding ---
secure_doc = EncryptedReport()
print(secure_doc.generate())      # Output: [ENCRYPTED] Base Report Content
```

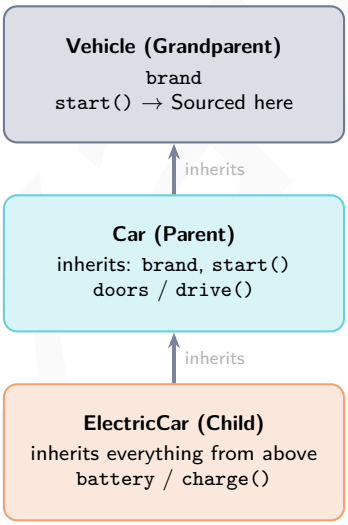
6.3.4 Multilevel Inheritance

Multilevel Inheritance — Core Concepts

Multilevel inheritance occurs when a child class inherits from a parent class, which in turn inherits from another parent class, creating a linear, multi-tiered chain (**Grandparent** → **Parent** → **Child**).

- **Linear Accumulation:** The lowest-level child class automatically inherits all attributes and methods across every structural tier above it.
- **The super() Forwarder:** Using `super().__init__()` allows constructors to pass arguments cleanly up to the immediate parent layer without explicitly hardcoding class names.
- **One-Way Visibility:** Lower classes can seamlessly see and call methods from higher-up tiers, but higher classes have absolutely no visibility down the chain.

Think of it like a vehicle manufacturing line: a basic `Vehicle` establishes tracking, a `Car` adds doors, and an `ElectricCar` adds battery specifications.



Example 147: Three-Tier Linear Inheritance Chain

```python
class Vehicle:
    """Tier 1: The Grandparent Class"""
    def __init__(self, brand):
        self.brand = brand

    def start(self):
        return f"{self.brand} engine started."

class Car(Vehicle):
    """Tier 2: The Parent Class"""
    def __init__(self, brand, doors):
        super().__init__(brand) # Forward to Vehicle constructor
        self.doors = doors

    def drive(self):
        return f"Driving the {self.brand} car with {self.doors} doors."

```

```python
class ElectricCar(Car):
    """Tier 3: The Child Class"""
    def __init__(self, brand, doors, battery_kwh):
        super().__init__(brand, doors) # Forward to Car constructor
        self.battery_kwh = battery_kwh

    def charge(self):
        return f"Charging the {self.battery_kwh}kWh battery pack."

# --- Testing the Multi-Tier Blueprint ---
tesla = ElectricCar(brand="Tesla", doors=4, battery_kwh=75)

# Accessing methods accumulated from all 3 structural tiers
print(tesla.start())    # Tier 1 Grandparent Method
print(tesla.drive())    # Tier 2 Parent Method
print(tesla.charge())   # Tier 3 Child Method

# --- Verifying Method Resolution Order (MRO) ---
print([cls.__name__ for cls in ElectricCar.__mro__])
# Output: ['ElectricCar', 'Car', 'Vehicle', 'object']
```

6.3.5 Multiple Inheritance and MRO

6.3.5.1 Getting from Two Parents, Diamond Problem

Multiple Inheritance and MRO

Python allows a class to inherit from **more than one parent**:

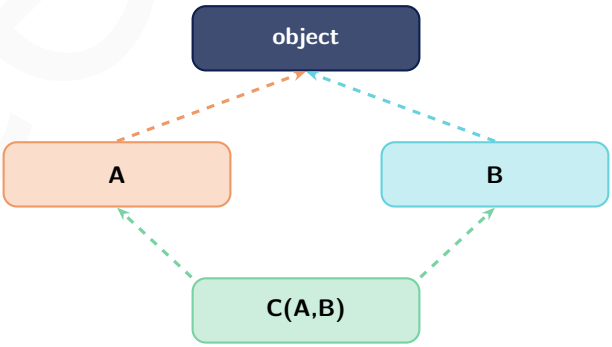
```
class C(A, B):
```

The diamond problem: if both `A` and `B` have a method with the same name, which one does `C` use?

Python uses the **MRO (Method Resolution Order)** to decide. The MRO is computed using the **C3 algorithm** (left-to-right, no repeats).

Simple rule: Python searches left-to-right through the parents listed in the class definition, going through each parent's own parents. `super()` follows this same order.

Check the MRO with `ClassName.__mro__` or `ClassName.mro()`.

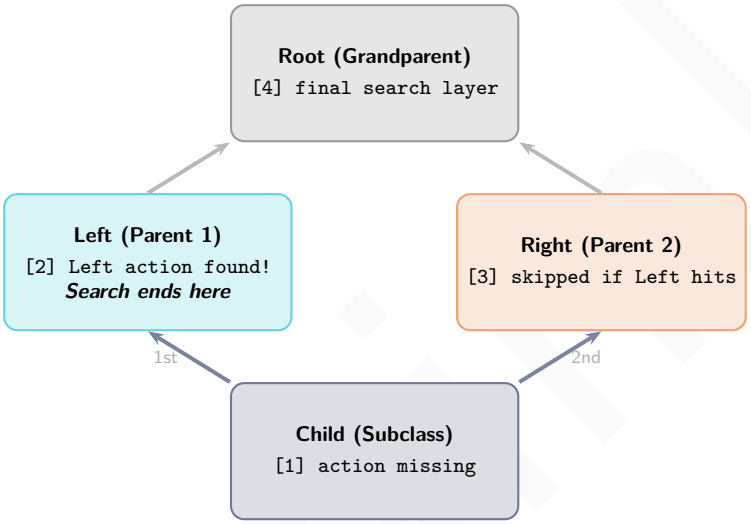


Multiple Inheritance — MRO Search Order

When a class inherits from multiple parents, Python determines method execution using the **C3 Linearization / MRO (Method Resolution Order)** algorithm. It follows strict layout principles:

- **Left-to-Right Priority:** For a class defined as `Child(Left, Right)`, Python evaluates the immediate parents from left to right.
- **Subclass First Rule:** A parent class (`Right`) will always be searched before a grandparent class (`Root`), even if `Left` is a direct child of `Root`.
- **Early Termination:** The search stops immediately at the first class that provides an implementation of the requested method.

Think of it as a depth-first search that is intentionally blocked from hitting the very top root until all local, lower-level subclass branches have been fully exhausted.



Example 148: Verifying the Python MRO Lookup Flow

```python
class Root:
    def action(self):
        return "Root action"

class Left(Root):
    def action(self):
        return "Left action"  # Python hits this first and stops

class Right(Root):
    def action(self):
        return "Right action" # Saved for step 3 if Left didn't implement it

class Child(Left, Right):
    pass # Child doesn't define the action, forcing an MRO search

# --- Execution Output ---
obj = Child()
print(obj.action())  # Output: Left action

# --- Inspecting the True Resolution Order Path ---
# Python explicitly lists out the lookup sequence in the __mro__ attribute:
print([cls.__name__ for cls in Child.__mro__])
# Output: ['Child', 'Left', 'Right', 'Root', 'object']
```

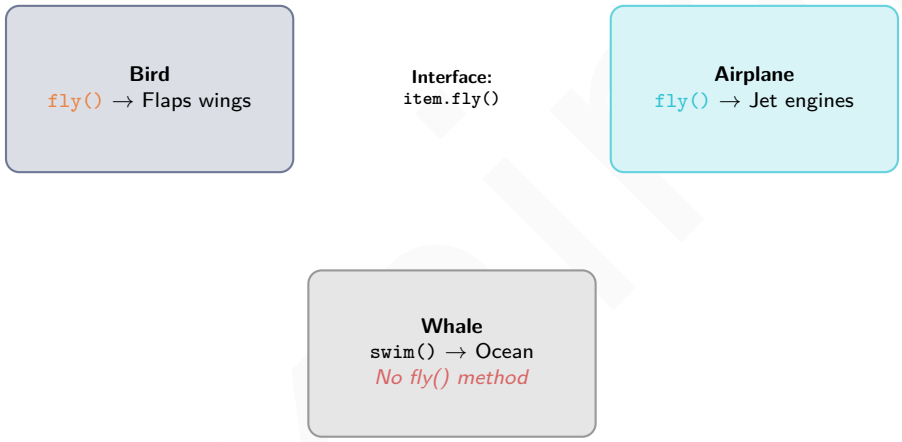
6.4 Polymorphism

Polymorphism — Core Concepts

Polymorphism literally means **"many forms."** In object-oriented programming, it allows different classes to be treated as if they were the same type through a shared, uniform interface.

- **Unified Interface:** It allows a single function, loop, or method to work seamlessly with different types of objects.
- **Customized Response:** Each individual object responds to the exact same method call in its own specific way.
- **Reduced Complexity:** It decouples your code from explicit type checks, letting you focus entirely on actions rather than underlying identity.

Think of it like an interface action: a `Car`, a `Plane`, and a `Boat` can all execute a `move()` action. The caller doesn't care *how* they move—the car drives, the plane flies, and the boat sails.



Polymorphism Types

- **Compile-Time (Overloading):** Python does not support native method overloading. It is simulated using default arguments (`None`) or variable arguments (`*args`).
- **Runtime (Overriding):** Subclasses redefine a parent method. The correct execution path is chosen dynamically at runtime based on the object instance.
- **Duck Typing:** Structural typing based on behaviors instead of inheritance. If an object has a required method at runtime, Python executes it without type-checking.
- **Operator Overloading**: Operator overloading allows the same operator to perform different actions depending on the objects it works with. Python supports operator overloading for both built-in data types and user-defined classes.

Example 149: Four Types of Pythonic Polymorphism

```
# 1. COMPILE-TIME OVERLOADING SIMULATION (Asset Depreciation)
class Asset:
    def depreciate(self, value, rate=None):
        if rate is not None:
            return value * rate        # Custom rate tracking
        return value * 0.10             # Standard default rate (10%)

# 2. RUNTIME METHOD OVERRIDING (Payment Processing Architecture)
class Payment:
```

```
    def pay(self): return "Standard processing pipeline"

class UPI(Payment):
    def pay(self): return "Redirecting to secure QR gateway"

class Wallet(Payment):
    def pay(self): return "Deducting tokens from application balance"

# 3. FUNCTION-BASED DUCK TYPING (File Export Subsystem)
class PDFExporter:
    def export(self): return "Writing byte streams to disk"

class CSVExporter:
    def export(self): return "Writing comma-separated strings"

def run_export_pipeline(engine):
    print(engine.export()) # No subclass dependency; checks for .export() method

# --- Execution Output Verification ---
# Overloading Check
asset = Asset()
print(asset.depreciate(1000))        # Output: 100.0
print(asset.depreciate(1000, 0.25))  # Output: 250.0

# Overriding Check
transactions = [UPI(), Wallet(), Payment()]
for txn in transactions:
    print(txn.pay())

# Duck Typing Check
run_export_pipeline(PDFExporter())   # Output: Writing byte streams to disk
run_export_pipeline(CSVExporter())   # Output: Writing comma-separated strings

# 4. BUILT-IN OPERATOR & FUNCTION POLYMORPHISM
print(100 + 50)              # Operator: Numeric addition (150)
print("Data" + "base")       # Operator: String concatenation (Database)
print(len("Python"))         # Function: Character count (6)
print(len([5, 10, 15]))      # Function: Item count (3)
print("GATE" * 3)            # GATEGATEGATE
```

Example 150: Polymorphism and Pythonic Duck Typing

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

# A polymorphic function that does not care about the underlying class type
def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!

# ------------------------------------------------------------
# Duck Typing: Focus on Behavior over Type Identity
# ------------------------------------------------------------
class Bird:
    def fly(self):
        return "Flapping wings"

class Airplane:
```

```
    def fly(self):
        return "Using jet engines"

# Both work inside the same loop because they both possess the .fly() method
entities = [Bird(), Airplane()]

for item in entities:
    print(item.fly())
# Output:
# Flapping wings
# Using jet engines
```

6.5 Encapsulation & Name Mangling

6.5.1 Hiding Data Inside a Class

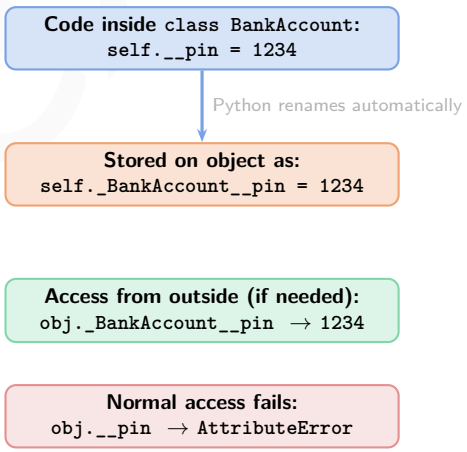
Encapsulation and Name Mangling

Encapsulation means keeping data inside a class and controlling how it is accessed or changed. Python uses **naming conventions** (not strict access control like Java's `private`):

Prefix	Meaning	Example
No prefix	Public, freely accessible	`self.name`
_ (one underscore)	"Please don't touch from outside"	`self._balance`
__ (two underscores)	Name mangling: harder to access by accident	`self.__pin`

Name mangling: `self.__pin` inside class `BankAccount` is stored as `self._BankAccount__pin`. This stops child classes from accidentally overwriting the same name.

Python is not truly private — anyone can still access `obj._BankAccount__pin` if they really want to. The double underscore is a *safety guard*, not a lock.



Example 151: Encapsulation — Bank Account Example

```
class BankAccount:
    def __init__(self, owner, balance, pin):
        self.owner    = owner          # public: ok to read
        self._balance = balance        # "please don't touch"
        self.__pin    = pin            # name mangled: hard to access

```

```python
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return f"Deposited {amount}. Balance: {self._balance}"

    def withdraw(self, amount, pin):
        if pin != self.__pin:
            return "Wrong PIN!"
        if amount > self._balance:
            return "Not enough money"
        self._balance -= amount
        return f"Withdrew {amount}. Balance: {self._balance}"

    @property
    def balance(self):                # read-only access via property
        return self._balance

acc = BankAccount("Alice", 10000, 1234)

# Public attribute: accessible
print(acc.owner)            # Alice

# Protected: accessible but convention says don't
print(acc._balance)         # 10000  (works but bad style)

# Name-mangled: acc.__pin fails
try:
    print(acc.__pin)        # AttributeError!
except AttributeError as e:
    print(e)                # 'BankAccount' object has no attribute '__pin'

# But you can still reach it this way (don't do this!)
print(acc._BankAccount__pin)   # 1234  (mangled name)

# Use the methods properly
print(acc.deposit(5000))       # Deposited 5000. Balance: 15000
print(acc.withdraw(2000, 9999)) # Wrong PIN!
print(acc.withdraw(2000, 1234)) # Withdrew 2000. Balance: 13000
print(acc.balance)              # 13000  (via property)

# Name mangling prevents child class collision
class SavingsAccount(BankAccount):
    def __init__(self, owner, balance, pin, interest_rate):
        super().__init__(owner, balance, pin)
        self.__pin = "child_pin"     # stored as _SavingsAccount__pin
                                     # does NOT overwrite BankAccount's __pin!

sa = SavingsAccount("Bob", 5000, 5678, 0.05)
print(sa.withdraw(1000, 5678))   # Withdrew 1000... (uses BankAccount's __pin)
```

6.6 Abstract Base Classes

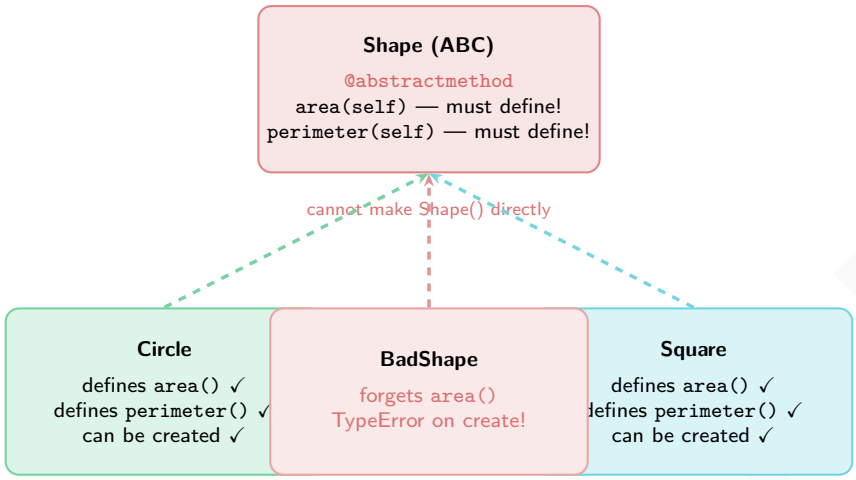
6.6.1 Forcing Child Classes to Implement Methods

Abstract Base Classes — What and Why

An **Abstract Base Class (ABC)** is a class that says: *"I am an incomplete template. Any class that inherits from me **must** define certain methods, or Python will not let you create objects from it."*

You mark methods as **abstract** with `@abstractmethod`. If a child class does not define all abstract methods, trying to create an object from it raises a `TypeError`.

Why use it: it is a **contract**. If you have many different classes (e.g. `Circle`, `Square`, `Triangle`) that should all have an `area()` method, define `Shape` as an ABC with abstract `area()`. Now Python enforces the contract.



Example 152: Abstract Base Class — Shape Example

```python
from abc import ABC, abstractmethod
import math

class Shape(ABC):   # ABC makes this an abstract class
    """Every shape must have area() and perimeter()."""

    @abstractmethod
    def area(self):
        pass   # no body needed -- child must provide it

    @abstractmethod
    def perimeter(self):
        pass

    # Non-abstract method: shared by all shapes
    def describe(self):
        return (f"{self.__class__.__name__}: "
                f"area={self.area():.2f}, "
                f"perimeter={self.perimeter():.2f}")

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):              # MUST define
        return math.pi * self.radius ** 2

    def perimeter(self):         # MUST define
        return 2 * math.pi * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width  = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Triangle(Shape):
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c

    def area(self):
        s = (self.a + self.b + self.c) / 2   # semi-perimeter
        return math.sqrt(s*(s-self.a)*(s-self.b)*(s-self.c))

```

```python
    def perimeter(self):
        return self.a + self.b + self.c

# Cannot create Shape directly
try:
    s = Shape()    # TypeError!
except TypeError as e:
    print(e)       # Can't instantiate abstract class Shape

# Can create concrete subclasses
c = Circle(5)
r = Rectangle(4, 6)
t = Triangle(3, 4, 5)

print(c.describe())    # Circle: area=78.54, perimeter=31.42
print(r.describe())    # Rectangle: area=24.00, perimeter=20.00
print(t.describe())    # Triangle: area=6.00, perimeter=12.00

# Use polymorphism: list of shapes
shapes = [Circle(3), Rectangle(2,5), Triangle(3,4,5)]
total_area = sum(s.area() for s in shapes)
print(f"Total area: {total_area:.2f}")    # Total area: 48.27

# Class that misses an abstract method
class BadShape(Shape):
    def area(self):
        return 0
    # forgot perimeter()!

try:
    b = BadShape()    # TypeError!
except TypeError as e:
    print(e)    # Can't instantiate abstract class BadShape
                # without an implementation for abstract method 'perimeter'
```

Example 153: OOP — Quick Reference

```python
# Q1: Class variable shared -- what prints?
class Counter:
    count = 0
    def __init__(self):
        Counter.count += 1

a = Counter()
b = Counter()
c = Counter()
print(Counter.count)    # 3  (shared across all instances)

# Q2: Instance variable shadows class variable
class Dog:
    legs = 4
d = Dog()
d.legs = 3              # creates instance var, shadows class var
print(d.legs)           # 3
print(Dog.legs)         # 4  (class var unchanged)

# Q3: MRO -- which method runs?
class A:
    def hello(self): return "A"
class B(A):
    def hello(self): return "B"
class C(A):
    def hello(self): return "C"
class D(B, C):
    pass

print(D().hello())      # B  (MRO: D -> B -> C -> A)
print(D.__mro__)        # D, B, C, A, object
```

```python

# Q4: super() chain
class X:
    def go(self): print("X")
class Y(X):
    def go(self): print("Y"); super().go()
class Z(Y):
    def go(self): print("Z"); super().go()
Z().go()   # Z / Y / X

# Q5: Property vs plain method
class Circle:
    def __init__(self, r): self.r = r
    @property
    def area(self): return 3.14 * self.r**2

c = Circle(5)
print(c.area)      # 78.5  (no parentheses -- it's a property)
# print(c.area())  # TypeError: 'float' is not callable
```

6.7 Problems

Problem 266 ***[MCQ]*** *What is the output?*

```
class Box:
    volume = 0

b1 = Box()
b2 = Box()
b1.volume = 10
print(b1.volume, b2.volume, Box.volume)
```

(A) `10 10 10`

(B) `10 0 0`

(C) `0 0 0`

(D) `10 10 0`

Problem 267 ***[MCQ]*** *What is the output?* ***(Mutable class variable trap)***

```
class Team:
    members = []

t1 = Team()
t2 = Team()
t1.members.append("Alice")
t2.members.append("Bob")
print(Team.members)
```

(A) `['Alice']`

(B) `['Bob']`

(C) `['Alice', 'Bob']`

(D) `[]`

Problem 268 ***[MCQ]*** *What is the output?*

```
class Counter:
    count = 0

    def __init__(self):
        Counter.count += 1

a = Counter()
b = Counter()
c = Counter()
print(Counter.count, a.count)
```

(A) `3 1`

(B) `3 3`

(C) `1 3`

(D) `1 1`

Problem 269 ***[MCQ]*** *What is the output?*

```
class Counter:
    count = 0

    def __init__(self):
        count += 1

a = Counter()
b = Counter()
c = Counter()
print(Counter.count, a.count)
```

(A) 3 1

(B) 3 3

(C) 1 3

(D) 1 1

Problem 270 ***[MCQ]*** *What is the output?*

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

a = A(5)
a.y = 99
print(a.x, a.y)
```

(A) 5 99

(B) 5 AttributeError

(C) AttributeError

(D) None 99

Problem 271 ***[MCQ]*** *What is the output?*

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

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

p = Point(3, 4)
print(p)
```

(A) <__main__.Point object>

(B) (3, 4)

(C) Point(3, 4)

(D) TypeError

Problem 272 ***[MCQ]*** *What is the output?*

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

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

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

print(Vector(1, 2) + Vector(3, 4))
```

(A) V(1,2)

(B) V(4,6)

(C) (4, 6)

(D) TypeError

Problem 273 ***[MCQ]*** *What is the output?*

```python
class Animal:
    def sound(self):
        return "..."

class Dog(Animal):
    def sound(self):
        return "Woof"

class Cat(Animal):
    pass

print(Dog().sound(), Cat().sound())
```

(A) `Woof Woof`

(B) `Woof ...`

(C) `... ...`

(D) `AttributeError`

Problem 274 ***[MCQ]*** *What is the output?*

```python
class Base:
    def __init__(self):
        self.x = 10

class Child(Base):
    def __init__(self):
        super().__init__()
        self.y = 20

c = Child()
print(c.x, c.y)
```

(A) `AttributeError: no attribute x`

(B) `10 20`

(C) `None 20`

(D) `10 None`

Problem 275 ***[MCQ]*** *What is the output?*

```python
class Base:
    def greet(self):
        return "Hello from Base"

class Child(Base):
    def greet(self):
        return super().greet() + " and Child"

print(Child().greet())
```

(A) `Hello from Base`

(B) `Hello from Base and Child`

(C) `and Child`

(D) `TypeError`

Problem 276 ***[MCQ]*** *What is the output?* ***(Multiple inheritance — MRO)***

```python
class A:
    def hello(self):
        return "A"

class B(A):
```

```
    def hello(self):
        return "B"

class C(A):
    def hello(self):
        return "C"

class D(B, C):
    pass

print(D().hello())
```

(A) A

(B) B

(C) C

(D) AttributeError

Problem 277 ***[MCQ]*** *What is the output?*

```
class A:
    def f(self):
        return "A"

class B(A):
    pass

class C(B):
    def f(self):
        return "C"

print(C().f(), B().f())
```

(A) C A

(B) A A

(C) C C

(D) AttributeError

Problem 278 ***[MCQ]*** *What is the output?*

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

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return 3.14 * self.r ** 2

c = Circle(5)
print(isinstance(c, Shape))
print(isinstance(c, Circle))
```

(A) False True

(B) True True

(C) True False

(D) False False

Problem 279 ***[MCQ]*** *What is the output?*

```python
class A:
    def __init__(self):
        self.x = 1

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 2

class C(B):
    def __init__(self):
        super().__init__()
        self.z = 3

obj = C()
print(obj.x, obj.y, obj.z)
```

(A) `AttributeError: x not found`

(B) `1 2 3`

(C) `None None 3`

(D) `1 2 None`

Problem 280 ***[MCQ]*** *What is the output?*

```python
class X:
    def greet(self):
        print("X", end=" ")

class Y(X):
    def greet(self):
        super().greet()
        print("Y", end=" ")

class Z(Y):
    def greet(self):
        super().greet()
        print("Z", end=" ")

Z().greet()
```

(A) `Z Y X`

(B) `X Y Z`

(C) `X Z Y`

(D) `Z X Y`

Problem 281 ***[MCQ]*** *What is the output?* ***(Method overriding / runtime polymorphism)***

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

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, b, h):
        self.b, self.h = b, h

    def area(self):
        return 0.5 * self.b * self.h

```

```
shapes = [Rectangle(4, 5), Triangle(6, 3)]
for s in shapes:
    print(s.area(), end=" ")
```

(A) `0 0`

(B) `20 9.0`

(C) `9 20`

(D) `20 0`

Problem 282 ***[MCQ]*** *What is the output?* ***(Duck typing)***

```
class Dog:
    def speak(self):
        return "Woof"

class Duck:
    def speak(self):
        return "Quack"

class Robot:
    def speak(self):
        return "Beep"

def make_noise(obj):
    print(obj.speak())

for animal in [Dog(), Duck(), Robot()]:
    make_noise(animal)
```

(A) `Woof Quack Beep` *(one per line)*

(B) `TypeError`*: expected Animal*

(C) `Woof` *only*

(D) `AttributeError`

Problem 283 ***[MCQ]*** *Python does not support traditional method overloading (same name, different parameter types). How is this typically handled?*

(A) By defining multiple methods with the same name; Python picks the last one.

(B) Using default arguments or `*args` *to handle multiple calling signatures.*

(C) Using `@overload` *from the standard library to enforce types.*

(D) Python raises `SyntaxError` *if two methods share the same name.*

Problem 284 ***[MCQ]*** *What is the output?*

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

    def get_balance(self):
        return self.__balance

a = Account(1000)
print(a.get_balance())
print(a.__balance)
```

(A) `1000` *then* `1000`

(B) `1000` *then* `AttributeError`

(C) `AttributeError` *for both*

(D) `None` *then* `AttributeError`

Problem 285 ***[MCQ]*** *After the following code, how can you access the mangled attribute from outside the class?*

```
class Foo:
    def __init__(self):
        self.__secret = 42
```

(A) `obj.__secret`

(B) `obj._secret`

(C) `obj._Foo__secret`

(D) *It is completely inaccessible from outside.*

Problem 286 ***[MCQ]*** *What is the output?*

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

s = Shape()
print("ok")
```

(A) `ok`

(B) `None`

(C) `TypeError: Can't instantiate abstract class`

(D) `NotImplementedError`

Problem 287 ***[MCQ]*** *What is the output?*

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return 3.14 * self.r ** 2

c = Circle(5)
print(round(c.area(), 2))
```

(A) `TypeError`

(B) `78.5`

(C) `25`

(D) `0`

Problem 288 ***[MCQ]*** *What is the output?*

```
class Stack:
    def __init__(self):
        self.__data = []

    def push(self, x):
        self.__data.append(x)

    def pop(self):
```

```
        return self.__data.pop()

    def __len__(self):
        return len(self.__data)

    def __repr__(self):
        return str(self.__data)

s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.pop(), len(s), s)
```

(A) 3 2 [1, 2]

(B) 1 2 [2, 3]

(C) 3 3 [1, 2, 3]

(D) 3 2 [3, 2, 1]

Problem 289 ***[NAT]*** *How many times is `__init__` called in total after the following code runs?*

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None

n1 = Node(1)
n2 = Node(2)
n3 = Node(3)
n1.next = n2
n2.next = n3
```

6.8 YouTube Links and QR Codes

Lecture	Details	YouTube Link	QR Code
25	Object-Oriented Programming	https://youtu.be/Dv6IIML57sI	
26	Inheritance in Python | Single, Multilevel, Multiple, super() & MRO	https://youtu.be/xzrm7RaZsn0	
27	Polymorphism | Compile, Runtime, Duck Typing & Operator Overloading	https://youtu.be/5E56HbGGtMY	
28	Encapsulation, Name Mangling & Abstract Classes	https://youtu.be/kc2bIEF5aUg	
29	Problem Solving on Object Oriented Programming (OOP)	https://youtu.be/mAOI0FqMSU8	

Chapter 7

GATE PYQs

GATEPYQ 1 ***[GATE DA 2024]*** *Consider the following Python code:*

```python
def count(child_dict, i):
    if i not in child_dict.keys():
        return 1
    ans = 1
    for j in child_dict[i]:
        ans += count(child_dict, j)
    return ans

child_dict = dict()
child_dict[0] = [1,2]
child_dict[1] = [3,4,5]
child_dict[2] = [6,7,8]
print(count(child_dict,0))
```

Which ONE of the following is the output of this code?

(A) 6
(B) 1
(C) 8
(D) 9

GATEPYQ 2 ***[GATE DA 2024]*** *Consider the following Python function:*

```python
def fun(D, s1, s2):
    if s1 < s2:
        D[s1], D[s2] = D[s2], D[s1]
        fun(D, s1+1, s2-1)
```

What does this Python function `fun()` do? Select the ONE appropriate option below.

(A) It finds the smallest element in D from index s1 to s2, both inclusive.
(B) It performs a merge sort in-place on this list D between indices s1 and s2, both inclusive.
(C) It reverses the list D between indices s1 and s2, both inclusive.
(D) It swaps the elements in D at indices s1 and s2, and leaves the remaining elements unchanged.

GATEPYQ 3 ***[GATE DA 2025]*** *Consider the following Python declarations of two lists.*

```python
A = [1,2,3]
B = [4,5,6]
```

Which one of the following statements results in `A = [1, 2, 3, 4, 5, 6]`?

(A) A.extend(B)
(B) A.append(B)

(C) A.update(B)
(D) A.insert(B)

GATEPYQ 4 ***[GATE DA 2025]*** *Consider the following Python code snippet.*

```
A = {"this", "that"}
B = {"that", "other"}
C = {"other", "this"}
while "other" in C:
    if "this" in A:
        A, B, C = A - B, B - C, C - A
    if "that" in B:
        A, B, C = C | A, A | B, B | C
```

When the above program is executed, at the end, which of the following sets contains "this"?
(A) Only A
(B) Only B
(C) Only C
(D) A, C

GATEPYQ 5 ***[GATE DA 2025]*** *Consider the following Python code snippet.*

```
def f(a,b):
    if (a==0):
        return b
    if (a%2==1):
        return 2*f((a-1)/2,b)
    return b+f(a-1,b)
print(f(15,10))
```

The value printed by the code snippet is (Answer in integer)

GATEPYQ 6 ***[GATE DA 2026]*** *Consider the given Python program.*

```
def append_to_lst(val, lst=[]):
    lst.append(val)
    return lst

print(append_to_lst(1))
print(append_to_lst(2))
print(append_to_lst(3, []))
```

Which of the following is the correct output of this program?
(A) [1] [2] [3]
(B) [1] [1, 2] [3]
(C) [1] [2] [1, 2, 3]
(D) [1] [1, 2] [1, 3]

GATEPYQ 7 ***[GATE DA 2026]*** *A recursive function in Python is given.*

```
def mystery(n):
    if n <= 0:
        return 1
    else:
        return mystery(n-1) + mystery(n-2)
```

Now, consider the following function call: $mystery(4)$

Assume that a typical runtime stack is used to manage function calls. Each function call is pushed onto the stack and removed only after it finishes execution. Which of the following options denotes the total number of function calls (i.e., the total number of stack activations), including the initial call, to compute `mystery(4)`*?*
(A) 5
(B) 9
(C) 15
(D) 17

GATEPYQ 8 ***[GATE DA 2026]*** *Consider the given Python program.*

```python
def outer():
    x = []
    def inner(val):
        x.append(val)
        return x
    return inner

f1 = outer()
f2 = outer()
print(f1(10)) # Line P
print(f1(20)) # Line Q
print(f2(30)) # Line R
print(f1(40)) # Line S
```

Which of the following options is/are correct?

(A) f1 and f2 share the same list x
(B) Output of Line Q is [10, 20]
(C) Output of Line R is [10, 20, 30]
(D) Output of Line S is [10, 20, 40]

GATEPYQ 9 ***[GATE DA 2026]*** *Consider the given Python program.*

```python
def fun(L, i=0):
    if i >= len(L)-1:
        return 0
    if L[i] > L[i+1]:
        L[i+1], L[i] = L[i], L[i+1]
        return 1+fun(L, i+1)
    else:
        return fun(L, i+1)

data = [5, 3, 4, 1, 2]
count = 0
for _ in range(len(data)):
    count += fun(data)
print(count)
```

The output of the program is _____. (Answer in integer)

Lecture	Details	YouTube Link	QR Code
30	Solutions to GATE PYQs	https://youtu.be/3JdV3IeClFA	

Chapter 8

Solutions to Problems

Problems Covered	YouTube Link	QR Code
Problem 1 to 52 (Lec 5)	https://youtu.be/xRmlSKYti8Y	
Problem 53–128 (Lec 11)	https://youtu.be/TzahIwnXL2U	
Problem 129–174 (Lec 14)	https://youtu.be/M5DRn3qAkP4	

Problem 175–237 (Lec 22)	https://youtu.be/xVWfcSaqhBE	
Problem 238–265 (Lec 24)	https://youtu.be/gdQv1SrJgq4	
Problem 266–289 (Lec 29)	https://youtu.be/mAOI0FqMSU8	
Solutions to GATE PYQs (Lec 30)	https://youtu.be/3JdV3IeClFA	

Bibliography

[1] E. Matthes, *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, 3rd ed., No Starch Press, 2023.

[2] A. Sweigart, *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*, 2nd ed., No Starch Press, 2019.

[3] M. Lutz, *Learning Python*, 5th ed., O'Reilly Media, 2013.

[4] Z. A. Shaw, *Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code*, Addison-Wesley, 2017.

[5] A. B. Downey, *Think Python: How to Think Like a Computer Scientist*, 3rd ed., O'Reilly Media, 2024.

GateXAIML

Free GATE resources for Data Science & AI and CSE
Website: www.gatexaiml.in
Email: contact@gatexaiml.in