NAMASTE PYTHON

Complete Deep-Dive Notes

How Python Works Behind the Scenes

25 Episodes • Internals • OOP • Concurrency • Best Practices

In the style of Namaste JavaScript by Akshay Saini

Python 3.10+ • CPython Internals • Interview Preparation

PART 1: HOW PYTHON WORKS BEHIND THE SCENES

Episode 1: How Python Works & Execution Model

⚡ INTERVIEW FAVOURITE

What Happens When You Run python script.py?

Python is NOT compiled to machine code like C/C++. It is a high-level, interpreted language. But behind the scenes, there IS a compilation step:

1. Your .py source code is read by the Python interpreter (CPython by default).
2. The source code is COMPILED into bytecode (.pyc files stored in __pycache__/).
3. The bytecode is executed by the Python Virtual Machine (PVM), line by line.

So Python is both compiled AND interpreted: compiled to bytecode first, then the PVM interprets that bytecode.

Bytecode & the PVM

Bytecode: An intermediate, platform-independent representation of your source code. Not machine code. Inspect with the dis module.

PVM (Python Virtual Machine): The runtime engine that executes bytecode. Handles loading, executing instructions, memory management, and error handling.

```
import dis
def add(a, b): return a + b
dis.dis(add)
# Output: LOAD_FAST, LOAD_FAST, BINARY_ADD, RETURN_VALUE
```

CPython, PyPy, Jython

- **CPython:** Default implementation. Written in C. Reference counting + GC. Has the GIL.
- **PyPy:** JIT compiler. 10-100x faster for certain workloads. Still has the GIL.
- **Jython / IronPython:** Python for JVM/.NET. No GIL.

Python is Dynamically Typed

Variables are just names (labels) pointing to objects. The type is determined at runtime.

```
x = 10      # x points to an int object
x = 'hello' # now x points to a str object (no error!)
```

Summary

• Python: source code → bytecode (.pyc) → PVM executes bytecode
• CPython is the default interpreter; PyPy has JIT for speed
• Python is dynamically typed: variables are labels pointing to objects
• The PVM is the runtime engine; bytecode is platform-independent

---

Episode 2: Memory Management & Garbage Collection

⚡ INTERVIEW FAVOURITE

Everything is an Object

In Python, EVERYTHING is an object: integers, strings, functions, classes, modules. Every object has: identity (id()), type (type()), and value.

```
a = 42
print(id(a))    # Memory address
print(type(a))  # <class 'int'>
```

Reference Counting

Reference Counting: Every object has a counter tracking how many references point to it. When count drops to zero, memory is freed immediately.

```
import sys
a = [1, 2, 3]
print(sys.getrefcount(a))  # 2 (a + function argument)
b = a  # refcount increases
del b  # refcount decreases
```

Garbage Collection (Cyclic References)

Reference counting cannot handle circular references. Python's gc module detects and cleans cycles using a generational algorithm.

Integer Caching (Interning)

⚡ INTERVIEW FAVOURITE

CPython caches small integers from -5 to 256. All variables pointing to 42 point to the SAME object.

```
a = 256; b = 256
print(a is b)  # True (same object, cached)
a = 257; b = 257
print(a is b)  # False (different objects)
```

== vs is

⚡ INTERVIEW FAVOURITE

- **== (Equality):** Compares VALUES.
- **is (Identity):** Compares IDENTITY (same object in memory).

```
a = [1, 2]; b = [1, 2]
print(a == b)  # True  (same values)
print(a is b)  # False (different objects)
```

✅ BEST PRACTICE

Always use == for values. Only use 'is' for None checks: if x is None.

Summary

• Everything is an object with id, type, and value
• Reference counting = primary GC; cyclic GC handles circular references
• Small integers (-5 to 256) are cached by CPython
• == compares values; is compares identity (same object in memory)

Episode 3: Mutable vs Immutable Objects

⚡ INTERVIEW FAVOURITE

The Core Distinction

Immutable: Cannot change after creation. Modification creates a NEW object. Examples: int, float, str, tuple, frozenset, bool.

Mutable: Can change in-place. Same memory address. Examples: list, dict, set, bytearray.

Demonstration

```
# Immutable: string
s = 'hello'; print(id(s))
s = s + ' world'; print(id(s))  # DIFFERENT id!

# Mutable: list
lst = [1, 2, 3]; print(id(lst))
lst.append(4); print(id(lst))    # SAME id!
```

The Mutable Default Argument Trap

⚡ INTERVIEW FAVOURITE

```
# WRONG
def add_item(item, lst=[]):
    lst.append(item); return lst
print(add_item(1))  # [1]
```

```
print(add_item(2))  # [1, 2] NOT [2]!

# CORRECT
def add_item(item, lst=None):
    if lst is None: lst = []
    lst.append(item); return lst
```

Shallow vs Deep Copy

- **Shallow:** New object, same nested references. copy.copy() or lst[:].
- **Deep:** New object + recursively copies nested objects. copy.deepcopy().

Summary

• Immutable: int, str, tuple — new object on modification
• Mutable: list, dict, set — changed in-place
• Never use mutable default arguments; use None + check
• Shallow copy copies references; deep copy clones everything

---

Episode 4: Variables, Scope & the LEGB Rule

⚡ INTERVIEW FAVOURITE

Variables Are Labels

A variable is a NAME (label) that REFERS to an object. Multiple names can point to the same object.

```
a = [1, 2, 3]; b = a
b.append(4); print(a)  # [1, 2, 3, 4]
```

The LEGB Rule

⚡ INTERVIEW FAVOURITE

LEGB: The order Python searches for a variable:

- **L — Local:** Inside the current function.
- **E — Enclosing:** Inside any enclosing (outer) functions.
- **G — Global:** At the module/script level.
- **B — Built-in:** Python's built-in names (print, len, range).

global and nonlocal Keywords

- **global:** Access/modify module-level variable inside a function.
- **nonlocal:** Access/modify enclosing scope variable in nested function.

Summary

• Variables are labels pointing to objects, not containers
• LEGB: Local → Enclosing → Global → Built-in
• global for module-level; nonlocal for enclosing scope
• Multiple names can alias the same object

Episode 5: Data Types & Data Structures

⚡ **INTERVIEW FAVOURITE**

Built-in Data Types

Type	Examples	Mutable?	Ordered?
int, float, complex	42, 3.14, 2+3j	Immutable	—
str	'hello'	Immutable	Yes
list	[1, 2, 3]	Mutable	Yes
tuple	(1, 2, 3)	Immutable	Yes
dict	{a: 1}	Mutable	Insertion (3.7+)
set	{1, 2, 3}	Mutable	No
NoneType	None	Immutable	—

List vs Tuple vs Set vs Dict

- **List:** Ordered, mutable, allows duplicates. O(1) append, O(n) search.
- **Tuple:** Ordered, immutable. Faster than lists. Hashable (can be dict keys).
- **Set:** Unordered, no duplicates. O(1) membership testing.
- **Dict:** Key-value pairs. O(1) lookup. Keys must be hashable.

Summary

• 7 core structures: int/float/str/list/tuple/dict/set
• Lists: mutable+ordered; Tuples: immutable+ordered
• Sets: unique elements, O(1) lookup; Dicts: key-value, O(1) by key
• Use ''.join() for string building, not += in loops

Episode 6: Functions — First-Class Citizens

⚡ INTERVIEW FAVOURITE

Functions Are Objects

Functions can be assigned to variables, passed as arguments, returned from functions, and stored in data structures.

*args and **kwargs

⚡ INTERVIEW FAVOURITE

- ***args:** Extra positional arguments as a tuple.
- ****kwargs:** Extra keyword arguments as a dictionary.

```
def func(*args, **kwargs):
    print(args)    # (1, 2, 3)
    print(kwargs)  # {'name': 'Sudip'}
```

Pass by Object Reference

⚡ INTERVIEW FAVOURITE

Python is pass-by-object-reference. Mutable objects can be modified inside functions; immutable cannot.

Lambda Functions

```
square = lambda x: x ** 2
sorted_list = sorted(users, key=lambda u: u['age'])
```

Summary

• Functions are first-class objects
• *args = tuple; **kwargs = dict
• Pass by object reference: mutable modified in-place, immutable not
• Lambda = anonymous single-expression function

Episode 7: Closures

⚡ INTERVIEW FAVOURITE

What is a Closure?

Closure: A function that retains access to variables from its enclosing scope, even after the enclosing function has finished.

```
def outer(msg):
    def inner(): print(msg)
    return inner
```

```
hello = outer('Hello!'); hello()  # 'Hello!'
```

Function Factories

```
def multiplier(n):
    def multiply(x): return x * n
    return multiply
double = multiplier(2); print(double(5))  # 10
```

Summary

• Closure = inner function + enclosing scope variables
• Remembers enclosing variables even after outer function returns
• Foundation for decorators, function factories, and data encapsulation

---

Episode 8: Decorators

⚡ **INTERVIEW FAVOURITE**

What is a Decorator?

Decorator: A function that wraps another function to add behavior WITHOUT changing its code. Uses closures.

```
from functools import wraps
def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        import time; start = time.time()
        result = func(*args, **kwargs)
        print(f'{func.__name__} took {time.time()-start:.4f}s')
        return result
    return wrapper

@timer
def slow_func(): import time; time.sleep(1)
```

Decorators with Arguments

```
def repeat(n):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(n): result = func(*args, **kwargs)
            return result
        return wrapper
```

```
    return decorator
@repeat(3)
def say_hello(): print('Hello!')
```

Built-in Decorators

- **@staticmethod, @classmethod, @property:** Class-level decorators.
- **@functools.lru_cache:** Memoization—caches function results.

Summary

• Decorator = function wrapping another to add behavior
• @ syntax: @dec def f() equals f = dec(f)
• Always use @functools.wraps to preserve metadata
• Three nesting levels for decorators with arguments

PART 2: ITERATION, COMPREHENSIONS & GENERATORS

Episode 9: Iterators & the Iterator Protocol

⚡ **INTERVIEW FAVOURITE**

Iterable vs Iterator

- **Iterable:** Has __iter__() which returns an iterator. (list, str, dict, file)
- **Iterator:** Has __iter__() + __next__(). Produces values one at a time.

```
my_list = [1, 2, 3]
it = iter(my_list)
print(next(it))  # 1, 2, 3, then StopIteration
```

A for loop calls iter() then next() until StopIteration.

Summary

• Iterable has __iter__(); Iterator has __iter__() + __next__()
• for loop = syntactic sugar for iter() + next() + StopIteration
• Iterators are lazy and can only be traversed once

Episode 10: Generators

⚡ INTERVIEW FAVOURITE

What is a Generator?

Generator: Function using yield. Produces values lazily, suspending state between calls.

```
def countdown(n):
    while n > 0: yield n; n -= 1
for num in countdown(5): print(num)  # 5,4,3,2,1
```

yield vs return

- **return:** Terminates function, sends value.
- **yield:** Pauses function, sends value, resumes on next next() call.

Generator Expressions

```
squares = (x**2 for x in range(1000000))  # ~120 bytes vs ~8MB for list
```

Summary

• Generator = function with yield; lazy evaluation
• yield pauses and resumes; return terminates
• Generator expressions: (expr for x in iter) — near-zero memory
• Use for large datasets, pipelines, infinite sequences

Episode 11: Comprehensions

List Comprehension

```
[x**2 for x in range(10)]  # [0,1,4,9,...,81]
[x for x in range(20) if x % 2 == 0]  # evens
```

Dict Comprehension

```
{w: len(w) for w in ['hello', 'world']}
```

Set Comprehension

```
{len(w) for w in ['hello', 'hi', 'hey']}  # {2, 3, 5}
```

Nested

```
flat = [num for row in matrix for num in row]
```

✅ **BEST PRACTICE**

Comprehensions are faster and Pythonic. But keep them readable.

Summary

• List: [expr for x in iter if cond]; Dict: {k:v for x in iter}
• Faster than for-loop equivalents; Pythonic
• No tuple comprehension—() creates a generator
• Keep readable; if > 2 lines, use a regular loop

Episode 12: map, filter, reduce

```
nums = [1, 2, 3, 4]
list(map(lambda x: x**2, nums))      # [1, 4, 9, 16]
list(filter(lambda x: x%2==0, nums)) # [2, 4]
from functools import reduce
reduce(lambda a, b: a+b, nums)        # 10
```

✅ **BEST PRACTICE**

Comprehensions are generally preferred over map/filter for readability.

Summary

• map applies func to each element; filter keeps where True
• reduce accumulates to single value (functools)
• Comprehensions preferred for readability in most cases

PART 3: OBJECT-ORIENTED PYTHON

Episode 13: OOP — Classes, Objects & __init__

⚡ **INTERVIEW FAVOURITE**

Classes and Objects

```
class Dog:
    species = 'Canine'  # Class variable
```

```
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age
    def bark(self): return f'{self.name} says Woof!'
buddy = Dog('Buddy', 5)
```

self

self: First parameter of every method. Refers to current instance.

Class vs Instance Variables

- **Class variables:** Shared by all instances.
- **Instance variables:** Unique per object (defined with self.xxx).

Summary

• Class = blueprint; Object = instance
• __init__ = constructor; self = current instance
• Class variables shared; instance variables unique

Episode 14: Inheritance, Polymorphism & MRO

⚡ INTERVIEW FAVOURITE

```
class Animal: def speak(self): return 'Sound'
class Dog(Animal): def speak(self): return 'Woof!'
```

Types

- **Single, Multiple, Multilevel:** Python supports all three.

MRO

⚡ INTERVIEW FAVOURITE

C3 Linearization determines method search order. Check with ClassName.__mro__.

super()

Calls parent class method. Essential in __init__.

Duck Typing

Duck Typing: Python cares about what an object CAN DO, not what it IS.

Summary

• Inheritance: child extends parent; super() calls parent methods
• MRO (C3 linearization) resolves multiple inheritance order
• Duck typing: behavior over type
• Polymorphism: same interface, different implementations

Episode 15: Magic Methods (Dunder Methods)

⚡ **INTERVIEW FAVOURITE**

- **__init__:** Constructor.
- **__str__:** User-readable string (print).
- **__repr__:** Developer string (debugging).
- **__eq__, __lt__, __gt__:** Comparison operators.
- **__add__, __mul__:** Arithmetic operators.
- **__len__, __getitem__:** Container protocol.
- **__enter__, __exit__:** Context manager.
- **__call__:** Make instance callable.

Summary

• Dunder methods customize object behavior with built-in operations
• __str__ for users; __repr__ for developers
• __eq__/__add__ enable operator overloading
• __enter__/__exit__ for context managers

Episode 16: @classmethod, @staticmethod & @property

- **@classmethod:** Receives cls. Factory methods.
- **@staticmethod:** No self or cls. Utility functions.
- **@property:** Method as read-only attribute.

```
class Circle:
    def __init__(self, r): self._r = r
    @property
    def area(self): return 3.14 * self._r**2
    @classmethod
    def from_diameter(cls, d): return cls(d/2)
    @staticmethod
    def is_valid(r): return r > 0
```

Summary

- @classmethod: receives cls, factory methods
- @staticmethod: no self/cls, utility function
- @property: method as attribute (getter/setter)

PART 4: ADVANCED PYTHON & INTERNALS

Episode 17: Error Handling

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f'Error: {e}')
else:
    print('No errors!')
finally:
    print('Always runs!')
```

Custom Exceptions

```
class InsufficientFunds(Exception): pass
```

✅ **BEST PRACTICE**

Catch specific exceptions. Use else for success code. Use finally for cleanup.

Summary

- try/except/else/finally for structured error handling
- Catch specific exceptions; avoid bare except:
- Custom exceptions by subclassing Exception

Episode 18: Context Managers

Context Manager: __enter__ and __exit__ methods for the with statement. Ensures cleanup.

```
with open('file.txt') as f: content = f.read()
```

Custom with @contextmanager

```
from contextlib import contextmanager
@contextmanager
def timer():
    import time; start = time.time()
    yield
    print(f'Elapsed: {time.time()-start:.4f}s')
```

Summary

• with ensures resource cleanup even on exceptions
• __enter__ acquires; __exit__ releases
• @contextmanager simplifies creation with yield

Episode 19: The GIL

⚡ **INTERVIEW FAVOURITE**

GIL: Mutex in CPython: only ONE thread executes bytecode at a time.

Impact

- **CPU-bound:** Threading = no speedup. Use multiprocessing.
- **I/O-bound:** Threading works (GIL released during I/O).

Solutions

- **multiprocessing:** Separate processes, separate GILs, true parallelism.
- **asyncio:** Single-thread concurrency for I/O.
- **C extensions:** NumPy releases GIL for parallel C code.

Summary

• GIL = one thread executes Python bytecode at a time
• CPU-bound: multiprocessing; I/O-bound: threading/asyncio
• GIL protects reference counting; may become optional (PEP 703)

Episode 20: Concurrency

Model	Best For	GIL Impact
threading	I/O-bound	GIL limits CPU
multiprocessing	CPU-bound	Bypasses GIL

asyncio	High-concurrency I/O	No issue (1 thread)

asyncio Example

```
import asyncio
async def fetch(url):
    await asyncio.sleep(1); return f'Data from {url}'
async def main():
    results = await asyncio.gather(fetch('a'), fetch('b'))
asyncio.run(main())
```

Summary

• threading for I/O; multiprocessing for CPU; asyncio for high-concurrency I/O
• async/await = cooperative multitasking, single thread
• multiprocessing = separate processes, true parallelism

Episode 21: Modules, Packages & Virtual Environments

✅ **BEST PRACTICE**

- **Module:** Single .py file.
- **Package:** Directory with __init__.py.

```
python -m venv myenv
source myenv/bin/activate
pip freeze > requirements.txt
```

✅ **BEST PRACTICE**

ALWAYS use virtual environments. Never install globally.

Summary

• Module = .py file; Package = directory with __init__.py
• Always use venv; pin deps with requirements.txt
• Use if __name__ == '__main__': for runnable modules

Episode 22: File I/O & Working with Data

```
with open('data.txt', 'r') as f: content = f.read()
with open('out.txt', 'w') as f: f.write('Hello!')
```

JSON & CSV

```
import json
json.dumps({'a': 1})  # dict → JSON string
json.loads('{"a": 1}')  # JSON string → dict
```

Summary

• Always use with open() for auto file closing
• json.dumps() serializes; json.loads() deserializes
• Use pathlib for modern file path handling

Episode 23: Type Hints & Modern Python

✅ **BEST PRACTICE**

```
def greet(name: str, times: int = 1) -> str:
    return (f'Hello, {name}! ' * times).strip()
```

Dataclasses

```
from dataclasses import dataclass
@dataclass
class User:
    name: str; age: int; email: str = ''
```

Summary

• Type hints are optional; not enforced at runtime
• Use for docs, IDE support, mypy
• @dataclass auto-generates __init__, __repr__, __eq__

Episode 24: Testing & Debugging

✅ **BEST PRACTICE**

```
# test_math.py
def test_add(): assert add(2,3) == 5
# Run: pytest test_math.py -v
```

Debugging

- **breakpoint():** Interactive debugger (Python 3.7+).
- **logging:** Production-grade. DEBUG/INFO/WARNING/ERROR/CRITICAL.

Summary

• pytest is the standard; assert for verification
• breakpoint() for debugging; logging for production
• Write tests from day one

Episode 25: Python Best Practices & The Zen of Python

✅ **BEST PRACTICE**

The Zen of Python (import this)

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Readability counts.
- Errors should never pass silently.

Top Best Practices

4. Follow PEP 8 (use Black/Ruff for formatting).
5. Use virtual environments for every project.
6. Use type hints for function signatures.
7. Prefer comprehensions over map/filter.
8. Use generators for large datasets.
9. Never use mutable default arguments.
10. Use context managers (with) for resources.
11. Catch specific exceptions.
12. Use f-strings for formatting.
13. Write tests from day one (pytest).
14. Use dataclasses for data classes.
15. Use enumerate() instead of range(len()).
16. Use logging instead of print() in production.
17. Profile before optimizing (cProfile, timeit).
18. Use pathlib over os.path.

Summary

• PEP 8, Black/Ruff, venv, type hints, tests = non-negotiable
• Generators for large data; comprehensions for readability

• The Zen: readability, simplicity, explicitness above all

FINAL CONSOLIDATED SECTION

Combined Summary

These notes cover Python from its execution model (bytecode, PVM, CPython) through memory management (reference counting, GC, interning), core features (mutable/immutable, LEGB, closures, decorators, generators, comprehensions, OOP, magic methods), to advanced topics (GIL, concurrency, context managers, type hints, testing) and best practices.

The approach mirrors Namaste JavaScript: understanding HOW Python works behind the scenes makes you a fundamentally better developer.

Top 20 Key Takeaways

19. Python compiles source to bytecode (.pyc), then the PVM executes it.
20. Everything is an object with id, type, and value; reference counting manages memory.
21. Mutable (list, dict, set) vs immutable (int, str, tuple) determines behavior.
22. LEGB rule governs scope: Local → Enclosing → Global → Built-in.
23. Functions are first-class objects: assign, pass, return them.
24. Closures = inner functions remembering enclosing scope.
25. Decorators wrap functions to add behavior; use @functools.wraps.
26. Generators use yield for lazy evaluation; near-zero memory.
27. Comprehensions are Pythonic and faster than for loops.
28. OOP: classes are blueprints; __init__ initializes; self = instance.
29. Dunder methods customize object behavior with built-in operations.
30. The GIL allows only one thread to execute bytecode at a time.
31. CPU-bound: multiprocessing; I/O-bound: threading/asyncio.
32. Context managers (with) ensure resource cleanup.
33. Never use mutable default arguments.
34. == compares values; is compares identity.
35. Type hints are documentation, not enforcement.
36. @dataclass auto-generates boilerplate.
37. pytest for testing; breakpoint() for debugging.

38. Follow PEP 8; use venv; write readable code.

Python Interview Cheat Sheet

Most commonly asked topics:

- How does Python execute code? (bytecode, PVM, CPython)
- Everything is an object. id(), type(), reference counting.
- Mutable vs Immutable: list vs tuple, mutable default arg trap.
- LEGB scope rule. global and nonlocal.
- *args and **kwargs. Pass by object reference.
- Closures: what, how, practical examples.
- Decorators: how they work, @wraps, with arguments.
- Generators: yield vs return, generator expressions.
- Comprehensions vs map/filter.
- OOP: __init__, self, class vs instance variables, inheritance.
- Magic methods: __str__ vs __repr__, __eq__, operator overloading.
- @classmethod vs @staticmethod vs @property.
- The GIL: what, why, CPU-bound vs I/O-bound.
- threading vs multiprocessing vs asyncio.
- == vs is. Integer caching / string interning.
- Context managers: with, __enter__/__exit__.
- Error handling: try/except/else/finally.
- Shallow vs deep copy.
- Type hints, dataclasses, f-strings.
- The Zen of Python. PEP 8. Best practices.